

SEO API Data Overview

 chatgpt.com/c/bf1e3c50-d75b-4539-a532-28a9442db5da

ChatGPT 4o



Hey ChatGPT, what can you tell me about the data for SEO API?



The data for an SEO API typically includes a variety of metrics and information that can help optimize a website's search engine performance. Some of the key data points an SEO API might provide include:

1. **Keyword Rankings:** Information on how specific keywords are ranking on search engines, along with historical data to track changes over time.
2. **Backlink Data:** Details about the backlinks pointing to a website, including the number, quality, and sources of these links.
3. **Site Audit Information:** Insights into on-page SEO factors, such as meta tags, content quality, site speed, mobile-friendliness, and technical SEO issues like broken links or crawl errors.
4. **Traffic Analytics:** Data on organic search traffic, including the volume of visits, bounce rate, and user behavior on the site.
5. **Competitor Analysis:** Information on competitors' SEO strategies, keyword rankings, and backlink profiles to identify opportunities and threats.
6. **Keyword Research:** Suggestions for new keywords, along with data on search volume, competition, and potential traffic.

This data can be used to make informed decisions about SEO strategies, optimize website content, and improve overall search engine rankings. Are you looking for information on a specific SEO API or trying to implement one for your website?

Sorry, misunderstood. I am talking about the product, or the SaaS, called, quote, Data4SEO, end quote.



Ah, got it! DataForSEO is a SaaS product providing a suite of APIs designed to help businesses with various aspects of SEO, SEM, and digital marketing. Here's an overview of what DataForSEO offers:

1. **Rank Tracking API:** Provides accurate and up-to-date data on keyword rankings across multiple search engines and locations.
2. **SERP API:** Allows users to access search engine results pages (SERPs) data, including organic and paid results, features snippets, and more.
3. **Keyword Data API:** Offers tools for keyword research, providing data on search volume, keyword difficulty, CPC, and related keywords.
4. **On-Page SEO API:** Delivers insights into on-page SEO factors, helping identify issues that can be optimized for better performance.
5. **Backlinks API:** Provides detailed information on backlinks, including metrics like domain authority, anchor text, and link status.
6. **Traffic Analytics API:** Offers data on website traffic sources, user behavior, and competitive traffic analysis.
7. **Google Trends API:** Enables users to analyze search trends over time to identify emerging opportunities and topics.

DataForSEO is widely used by digital marketers, SEO professionals, and agencies to enhance their data-driven strategies and improve search engine performance. Are you considering using it for your business?

Yes, actually, we have a very important problem that we're looking to solve. The problem is that we have clients whose Google Business profiles are being overtaken by competitors because competitors are able to generate reviews at a faster velocity than our clients are able to at times. We want to be able to monitor competitors and look at how many reviews they have, keep track of the number of reviews, and track the trend month over month for how many reviews these companies are acquiring on a month-to-month basis, and compare that to our clients' Google Business profiles and their review counts and review velocity. Which API would best help us to accomplish this?



For your needs, the **Business Data API** from DataForSEO would be a great fit. This API is designed to provide insights into Google Business Profiles and their associated data, including reviews. Here's how it can help you:

1. **Review Tracking:** Monitor the number of reviews on both your clients' and their competitors' Google Business profiles.
2. **Review Trend Analysis:** Analyze the review acquisition trends over time, allowing you to see how review counts are changing month-over-month.
3. **Competitor Analysis:** Compare your clients' review metrics against their competitors to identify areas where they need to improve.

By leveraging this API, you can keep a close eye on how competitors are performing in terms of review generation and adjust your strategies to help your clients maintain or regain their competitive edge.

That's great. Can you tell me something? We have a Looker Studio template that we set up for clients where we combine data sources from Google Analytics, Google Search Console, and Google My Business, and I'm wondering if it's possible to use an API like Data for SEO to query the competitors for a client and provide metrics inside of the dashboard for things like review count, review velocity, and things of that nature. Is it possible to integrate a third-party API somehow into Looker Studio so that we're displaying data there, possibly storing some data maybe in something like BigQuery in order to keep a historical record of what some of this data looks like, unless that is that Data for SEO is able to give us historical data without the need for storing information.



Yes, you can integrate third-party APIs like DataForSEO into Looker Studio to display metrics such as review count and review velocity alongside other data sources like Google Analytics, Google Search Console, and Google My Business. Here's a general approach to achieve this integration:

1. **API Integration:** Use a service like Google Cloud Functions or a server with scheduled tasks to call the DataForSEO API regularly. This service will gather data on review counts, review velocity, and other metrics for your clients and their competitors.
2. **Data Storage:** Store the fetched data in a database like Google BigQuery. This allows you to keep a historical record of the data, which is especially useful if the API doesn't provide historical data directly. BigQuery can store large datasets efficiently and is well-integrated with Looker Studio.

3. **Looker Studio Connection:** Connect Looker Studio to BigQuery to visualize the stored data. You can create custom charts and dashboards that display metrics like review count trends, review velocity, and comparisons between your clients and their competitors.
4. **Dashboard Customization:** Customize your Looker Studio dashboard to present the data in a way that's meaningful for your clients. This might include visualizations like time series charts for review trends, comparison tables, and alerts for significant changes in competitor metrics.

By using this approach, you can provide your clients with a comprehensive view of their Google Business performance compared to their competitors, leveraging the power of DataForSEO and Looker Studio.

Can you elaborate a little bit on what a server with scheduled tasks entails?



A server with scheduled tasks is a setup where a server or a cloud-based computing resource is used to run specific tasks at regular intervals. These tasks can include fetching data from APIs, processing data, and storing it in a database. Here's how it generally works:

1. **Server Setup:** You can use a physical server, a virtual machine, or a cloud service like AWS EC2, Google Cloud Compute Engine, or Azure Virtual Machines. These platforms provide the computing resources needed to run your scripts or applications.
2. **Task Scheduling:** Use a scheduling tool to automate when and how often your tasks run. Common tools for scheduling tasks include:
 - o **Cron Jobs:** On Unix-like systems, cron is used to schedule scripts or commands to run at specific times or intervals. You define the schedule using a cron expression.
 - o **Task Scheduler:** On Windows, Task Scheduler can be used to automate the running of tasks at scheduled times.
 - o **Cloud Scheduler:** Google Cloud Scheduler is a fully managed cron job service that allows you to run tasks at regular intervals without managing your own server.
3. **Script or Application:** Write a script or application that performs the desired task, such as calling an API, processing the returned data, and storing it in a database. This script could be written in languages like Python, JavaScript (Node.js), or any other language you're comfortable with.

4. **Data Storage:** Once the data is fetched and processed, it's stored in a database like Google BigQuery, SQL, or NoSQL databases. This stored data can then be used for analysis and visualization.
5. **Monitoring and Maintenance:** Ensure the server and scheduled tasks are monitored for errors or failures. Implement logging and alerting mechanisms to notify you of any issues that may arise.

Using a server with scheduled tasks allows you to automate data collection and processing, ensuring that you have up-to-date information available for analysis and reporting. This setup can be particularly useful when integrating multiple data sources into dashboards like those in Looker Studio.

It sounds like Google's cloud scheduler would be the easiest route. Would that be a fair assumption?



Yes, using Google Cloud Scheduler is often one of the easiest and most convenient routes, especially if you're already working within the Google Cloud ecosystem. Here's why it might be the best choice:

1. **Managed Service:** Google Cloud Scheduler is a fully managed service, meaning you don't have to worry about server maintenance or uptime. It handles the scheduling of tasks reliably without requiring you to manage infrastructure.
2. **Integration:** It integrates seamlessly with other Google Cloud services, such as Google Cloud Functions, App Engine, and Pub/Sub. This makes it easier to set up workflows that involve triggering functions or processes in your cloud environment.
3. **Ease of Use:** Setting up a task in Cloud Scheduler is straightforward. You can use a simple web interface or command-line tools to define your cron jobs, and it supports flexible scheduling expressions.
4. **Scalability:** Cloud Scheduler can handle tasks of various sizes and complexities, scaling to meet your needs as they grow.
5. **Security:** It offers built-in security features, allowing you to manage permissions and access controls using Google Cloud IAM, ensuring that only authorized users can create or modify tasks.

Using Google Cloud Scheduler can streamline your workflow by automating tasks with minimal overhead and maintenance, making it a practical choice for integrating with services like DataForSEO and storing data in BigQuery.

Can you elaborate a little bit on what Google Cloud Functions is?



Google Cloud Functions is a serverless computing service offered by Google Cloud Platform. It allows you to run your code in response to events without having to manage any server infrastructure. Here's a bit more about how it works and its benefits:

1. **Event-Driven:** Cloud Functions can be triggered by events from various sources, such as HTTP requests, changes in Cloud Storage, messages in Pub/Sub, or even events from other Google Cloud services.
2. **Serverless:** Being serverless means you don't have to provision or manage servers. You simply write your function code and deploy it. Google Cloud automatically handles the scaling and execution of your functions as needed.
3. **Flexible Language Support:** Cloud Functions supports several programming languages, including JavaScript (Node.js), Python, Go, Java, and others, making it versatile for developers.
4. **Scalability:** It automatically scales with the load. Whether you're handling one request or thousands, Cloud Functions scales up or down without requiring any configuration on your part.
5. **Pay-As-You-Go:** You only pay for the computing resources you use while your function is running. This cost-effective model is ideal for applications with variable or unpredictable workloads.
6. **Integration with Google Cloud Services:** Cloud Functions can easily integrate with other Google Cloud services, enabling you to build complex workflows. For example, you can trigger a Cloud Function when a new file is uploaded to Cloud Storage or when a message is published to Pub/Sub.
7. **Use Cases:** Common use cases include running backend services, responding to webhooks, processing files, handling HTTP requests, and automating tasks in response to cloud events.

Google Cloud Functions provides a powerful way to build and deploy lightweight, event-driven applications and services without the overhead of managing server infrastructure.

So can you just very simply clarify what the difference between Google Cloud Functions and Google Cloud Scheduler are?



Sure! The main difference between Google Cloud Functions and Google Cloud Scheduler lies in their purpose and how they operate:

1. Google Cloud Functions:

- **Purpose:** It's a serverless computing service designed to run your code in response to events.
- **Operation:** You write and deploy functions that execute automatically when triggered by specific events, such as an HTTP request, a file upload, or a message in a queue.
- **Use Case:** Ideal for running small pieces of code or scripts that respond to real-time events without managing any server infrastructure.

2. Google Cloud Scheduler:

- **Purpose:** It's a service for scheduling tasks, similar to cron jobs, that need to be executed at specific times or intervals.
- **Operation:** You schedule tasks that can trigger HTTP endpoints, Pub/Sub topics, or App Engine targets on a regular basis or at specific times.
- **Use Case:** Best for automating regular tasks or workflows, like running batch processes or performing regular data retrieval and processing tasks.

In summary, Cloud Functions is about executing code in response to events, while Cloud Scheduler is about scheduling tasks to run at specified times. They can be used together, for example, where Cloud Scheduler triggers a Cloud Function to perform a task at a scheduled interval.

Would it be fair to assume that we could implement a solution with data for SEO API by creating a Google Cloud Scheduler that runs or executes a Google Cloud Function code to get API data and store it to a BigQuery database and do that at set intervals so that the Cloud Scheduler manages the frequency of code execution and Google Cloud Function is managing the actual code that is being executed?



Yes, that's a fair assumption and a great way to leverage both Google Cloud Scheduler and Google Cloud Functions together. Here's how you can implement this solution:

1. Google Cloud Functions: Write a Cloud Function that:

- Calls the DataForSEO API to fetch the necessary data about reviews, competitors, etc.
- Processes this data as needed.
- Stores the processed data into a Google BigQuery table for further analysis and reporting.

2. **Google Cloud Scheduler:** Set up a Cloud Scheduler job that:

- Triggers the Cloud Function at specified intervals (e.g., daily, weekly, or monthly) based on your requirements.
- Uses HTTP requests to invoke the Cloud Function, passing any necessary parameters or configurations.

3. **BigQuery:** Use Google BigQuery to:

- Store the data retrieved by the Cloud Function, allowing for easy querying and integration with Looker Studio.
- Maintain a historical record of the data to analyze trends over time.

This setup allows you to automate the process of fetching, processing, and storing data at regular intervals without manual intervention, making it efficient and scalable for your needs.

Could you give me an approximation of what type of costs we would be looking at incurring by implementing this solution using Google Cloud Scheduler, Google Cloud Functions, and the data for SEO API for business information?



Estimating the costs for implementing a solution using Google Cloud Scheduler, Google Cloud Functions, and DataForSEO involves considering several factors, including the frequency of data retrieval, the volume of data, and the pricing model of each service. Here's a rough breakdown:

1. **Google Cloud Scheduler:**

Google Cloud Scheduler is quite cost-effective. As of now, the pricing is typically around a few cents per job invocation per month. If you're scheduling a job to run once a day, this might cost you just a few dollars per month.

2. **Google Cloud Functions:**

- Pricing is based on the number of function invocations, the duration of function execution, and the resources allocated (memory and CPU).
- For instance, you get two million invocations free per month. Beyond that, the cost depends on the memory allocated and execution time. A simple function running briefly could cost a few dollars a month, depending on usage.

3. Google BigQuery:

- Costs are primarily driven by the amount of data stored and the volume of data processed during queries.
- Storage costs are approximately \$0.02 per gigabyte per month.
- Querying costs about \$5 per terabyte of data processed. However, keeping your datasets well-optimized can help keep costs low.

4. DataForSEO API:

Pricing for DataForSEO APIs is typically based on the number of API requests or the data volume. Their pricing can vary, so checking their website for specific pricing tiers or contacting them for a custom quote might be beneficial. Depending on the usage, this could range from tens to hundreds of dollars monthly.

Overall, if you're using these services efficiently, you might expect a monthly cost ranging from a few tens to a few hundred dollars, primarily driven by the DataForSEO API and BigQuery querying costs. The Google Cloud components (Scheduler and Functions) are usually the less expensive part of this setup, especially at low to moderate usage levels.

So for this particular use case, I think that we would be running these tasks through cloud scheduler at a frequency of probably just once a month. Forgetting or ignoring the data for SEO API costs, can you give me a rough approximation of what the cost would be for the rest of the setup?



If you're running the tasks only once a month using Google Cloud Scheduler and Google Cloud Functions, the costs would likely be minimal. Here's a rough approximation:

1. Google Cloud Scheduler:

Since you plan to run the task once a month, the cost for Cloud Scheduler will be negligible, possibly just a few cents per month.

2. Google Cloud Functions:

- With infrequent invocations (once a month), you'll likely remain well within the free tier for invocations, which is two million per month.
- If your function runs for a short time (e.g., seconds) and uses minimal memory, the execution costs will also be minimal, likely less than a dollar per month.

3. Google BigQuery:

- **Storage Costs:** If your dataset is small (a few gigabytes), the storage cost will be very low, approximately a few cents to a couple of dollars per month.
- **Query Costs:** With a once-a-month querying schedule, the costs are likely to remain under a dollar unless you're processing very large datasets.

Overall, excluding the DataForSEO API costs, you're looking at a monthly expense that's likely under five dollars, primarily driven by minimal usage of Cloud Functions and BigQuery. This setup is highly cost-effective, especially at low usage levels.

Can you explain to me how I go about accessing Google Cloud Functions and Google Cloud Scheduler?



Accessing Google Cloud Functions and Google Cloud Scheduler involves setting up a Google Cloud Platform (GCP) account and using the Google Cloud Console to configure your services. Here's a step-by-step guide to get you started:

Setting Up Your Google Cloud Platform Account

1. **Create a GCP Account:** If you don't have one already, sign up for a Google Cloud Platform account at the Google Cloud website. New users often get a free trial with credits to explore various services.
2. **Set Up a Project:** After logging in to the Google Cloud Console, create a new project. This will be the environment where you manage and deploy your services.

Accessing Google Cloud Functions

1. Navigate to Cloud Functions:

In the Google Cloud Console, go to the **Navigation Menu** and select **Cloud Functions** under the **Compute** section.

2. Enable the API:

If prompted, enable the Cloud Functions API for your project.

3. Create a Cloud Function:

- Click on **Create Function**.
- Configure the basic settings, including the function name, region, and trigger type (e.g., HTTP trigger for easy testing).
- Write your function code directly in the console or upload a ZIP file with your code.
- Set any necessary environment variables or dependencies.

4. Deploy the Function:

Once configured, click **Deploy** to make your function live. This process might take a few minutes.

Accessing Google Cloud Scheduler

1. Navigate to Cloud Scheduler:

In the Google Cloud Console, go to the **Navigation Menu** and select **Cloud Scheduler** under the **Tools** section.

2. Enable the API:

If prompted, enable the Cloud Scheduler API for your project.

3. Create a Cloud Scheduler Job:

- Click on **Create Job**.
- Fill out the job details, including the name, frequency (using cron syntax for scheduling), and timezone.
- Set the **Target** to HTTP and provide the URL of the Cloud Function you want to trigger.
- Configure any necessary headers or authentication methods, if applicable.

4. Save and Test:

Save your job and test it to ensure it correctly triggers the Cloud Function.

Additional Setup

- **IAM Permissions:** Ensure that the necessary permissions are set for both Cloud Functions and Cloud Scheduler. This might involve setting roles for service accounts that allow them to invoke functions or access other services.
- **Monitoring and Logs:** Use Google Cloud Logging and Monitoring to track the performance and invocations of your functions and scheduled jobs, making debugging easier if issues arise.

By following these steps, you'll be able to set up and access both Google Cloud Functions and Google Cloud Scheduler, allowing you to automate tasks and manage serverless applications within your GCP environment.

Can you explain briefly what pub/sub is?



Google Cloud Pub/Sub is a messaging service that allows you to send and receive messages between independent applications or components in real time. It follows a publish/subscribe (pub/sub) model, which decouples the entities that send messages (publishers) from the entities that receive messages (subscribers). Here's a brief overview of how it works and its key features:

How Pub/Sub Works

1. Topics:

Publishers send messages to a topic. A topic is a named resource to which messages are sent by publishers.

2. Subscriptions:

Subscribers express interest in one or more topics by creating a subscription to a topic. A subscription represents the stream of messages from the topic to the subscribing application.

3. Messages:

Messages are sent by publishers to topics. These messages are then forwarded to all subscriptions of that topic.

4. Delivery:

Subscribers pull messages from the subscription, or they can receive messages via push delivery to an endpoint.

Key Features

- **Scalability:** Pub/Sub can handle large volumes of messages and can scale automatically to meet demand.
- **Asynchronous Communication:** Enables different parts of an application to communicate asynchronously, improving efficiency and decoupling components.
- **Reliability:** Provides at-least-once delivery of messages, ensuring that messages are not lost.
- **Flexible Delivery:** Supports both push and pull delivery mechanisms, allowing subscribers to choose how they want to receive messages.
- **Integration:** Easily integrates with other Google Cloud services, making it a powerful tool for building event-driven architectures.

Use Cases

- **Event Ingestion:** Collect and distribute events from various sources for real-time processing and analytics.
- **Decoupling Microservices:** Allow independent development and scaling of microservices that communicate through Pub/Sub.
- **Real-time Data Streaming:** Stream data for applications that require real-time updates, such as monitoring and alerting systems.

Google Cloud Pub/Sub is a versatile messaging service that simplifies communication between different parts of an application, enabling you to build more resilient and scalable systems.

Okay, can you help me plan the implementation in terms of BigQuery. Here's what I'm thinking the actions and functionality that the script will need to include: - Connect to a BigQuery database - Find a list of Customers in BigQuery (our agency's customers) - Get a list of queries in for each customer in BigQuery. Queries are stored in a comma separated list. - Use the DataForSEO API to perform all queries for a customer one by one. - For each query, the script will need to identify the top 5 competitors and the position of our customer. - After performing all of the queries for a customer the script must create a deduped list of all top competitors for all queries. - The script will need to populate the list of competitors in BigQuery and connect them to the customer. - For each competitor, the script will need to use the DataForSEO API to find key datapoints for the competitor including review count and review velocity (if possible) - The script must perform these actions for all customer records. I have attached a diagram of what I think is a possible set up for the BigQuery Database. Can you tell me if this database setup is possible with BigQuery, if its necessary to split the database into multiple tables like this, or if there is a better/simpler solution?



Your proposed database setup in BigQuery, as depicted in your diagram, is a sound approach for managing data relationships in a relational database style. Here's a breakdown of the setup and some considerations for using BigQuery effectively:

Analysis of Your Proposed Setup

1. Customers Table:

Contains basic customer information and a list of queries (comma-separated).

2. Customers Competitors Table (Join Table):

- Acts as a many-to-many relationship table between customers and competitors.
- Stores pairs of **Customer ID** and **Competitor ID**.

3. Competitors Table:

Stores competitor details including review count and review velocity.

Considerations and Recommendations

1. Use of Tables:

Your setup using multiple tables to represent relationships is valid and facilitates organized data management, especially for handling many-to-many relationships effectively in BigQuery.

2. Handling of Queries:

Storing queries as a comma-separated list in the Customers Table could be less efficient for processing. Consider storing each query in a separate table row linked to the customer ID for more straightforward processing and better scalability.

3. Schema Design:

- Ensure each table includes a primary key (e.g., **Customer ID** in Customers Table, **Competitor ID** in Competitors Table).
- Use indices on frequently accessed fields like **Customer ID** in the join table to optimize query performance.

4. Normalization:

Your approach to normalization by separating data into different tables helps in maintaining data integrity and reducing redundancy. However, BigQuery is optimized for handling large datasets with denormalized structures. Depending on your query patterns and data volume, consider if a denormalized schema might be more performance-efficient in some cases.

5. API Integration and Data Insertion:

- When integrating with the DataForSEO API and inserting data into BigQuery, ensure your scripts handle data type mapping correctly and manage any API limits or errors gracefully.
- Use batch inserts or streaming inserts to populate BigQuery tables efficiently.

6. Performance Considerations:

Since BigQuery is optimized for large-scale analytics, ensure that your queries are optimized for performance, especially when joining large datasets.

Implementation Steps Using Google Cloud Functions

Your script in a Google Cloud Function can be structured to:

- Connect to BigQuery.
- Retrieve customer data and associated queries.
- Loop through each query to fetch competitor data from DataForSEO.
- Deduplicate competitor data and populate the Competitors Table and Customers Competitors Table.
- Schedule this script using Google Cloud Scheduler to run at your desired frequency.

Example of a Simple Script Workflow

1. Fetch Customers:

Query BigQuery to get the list of customers.

2. Process Queries for Each Customer:

- Split the comma-separated list of queries.
- For each query, use DataForSEO to fetch competitors and update or insert data into the Competitors Table.

3. Update Relationships:

Ensure the Customers Competitors Table is updated with new relationships.

4. Performance Monitoring:

Regularly monitor the function's execution time and query costs in BigQuery to ensure the setup remains cost-effective.

This plan should provide a robust foundation for automating the competitor analysis for your clients using BigQuery and Google Cloud Platform services.

Is this a better database set up?



Yes, this revised setup you've outlined is an improvement for several reasons:

1. **Normalization:** By separating customer queries into a distinct table (**Customer Queries**), you improve the database's normalization, which helps reduce redundancy and improve data integrity. This structure makes it easier to manage queries independently of customer records.
2. **Clarity and Scalability:** The separation enables more straightforward scalability and updates. You can add, modify, or remove queries without impacting the main customer data. This structure is particularly beneficial for applications with a high volume of changes or additions to query data.
3. **Database Performance:** With this setup, queries on individual tables can be more efficient, especially if indexes are used effectively. For example, joining the **Customer Queries** table with the **Customers** table on **Customer ID** allows for efficient lookups and updates.
4. **Flexibility in Data Management:** This design offers more flexibility in handling data. For instance, if you need to update a query, you do not risk affecting customer data, and vice versa. It also makes it easier to handle permissions and access control on a more granular level.
5. **Simplified Data Manipulation and Reporting:** By isolating queries and competitor data, it becomes easier to write more straightforward and efficient SQL queries for reporting and data manipulation purposes. It can significantly enhance the performance of analytical queries and simplifies maintenance.

Overall, this layout is better suited for managing complex relationships and will likely result in improved performance and easier maintenance as your data scales. This model adheres more closely to best practices in database design, particularly for use cases involving relational data in analytical contexts like those seen with BigQuery.

I'm new to BigQuery. When I go to BigQuery Studio, I have an option to create a dataset. In BigQuery is a dataset the same as a database table?



In BigQuery, a **dataset** is not the same as a **database table**. Instead, it's helpful to think of it more like a database or schema in traditional relational databases. Here's a breakdown of how these concepts relate in BigQuery:

BigQuery Datasets

- **Definition:** A dataset in BigQuery is a container that holds tables, views, and other data objects. It helps organize and manage access to your data. Think of it as a database in a traditional SQL database system.

- **Purpose:** Datasets are used to organize data by project, subject area, or other categories. For example, you might have separate datasets for different departments within a company, like sales, marketing, and operations.
- **Access Control:** You can set access permissions at the dataset level, controlling who can access the tables and views contained within.

BigQuery Tables

- **Definition:** A table is where data is stored in BigQuery. Tables contain rows and columns, similar to tables in traditional relational databases.
- **Purpose:** Tables hold the actual data that you query with SQL in BigQuery. Each table has a defined schema that describes the data types and structure.
- **Types of Tables:** There are various types of tables in BigQuery, including native tables (where data is stored directly in BigQuery), external tables (where data can be stored in sources like Google Cloud Storage but queried through BigQuery), and materialized views (which store the results of a query for faster access).

Practical Example

If you're setting up your environment in BigQuery:

1. **Create a Dataset:** First, you create a dataset (akin to creating a database). This might be named after your project or the type of data it will contain, such as `CustomerData`.
2. **Create Tables:** Inside the dataset, you would create tables. For example, you might have tables like `Customers`, `Queries`, `Competitors`, etc., as per your database diagram.

Using BigQuery Studio

When you're in BigQuery Studio (also known as the BigQuery web UI):

- **Creating a Dataset:** This is like setting up a new database. You would typically do this once for a set of related data.
- **Creating Tables:** This is where you define the specifics of the data you will store, such as columns and their data types.

This structure helps keep your data organized and secure, while also allowing for efficient management and querying of large datasets.

I created a dataset called "CustomerGbpCompetitors". I then created a table called "Customers". The Customers table has the following fields: - CustomerID (Integer, Required) - CustomerName (String, 255 max length)
I have two questions: 1. How do I define the CustomerID as the primary

key for this table? 2. BigQuery has a table settings section called "Partition and cluster settings". A partitioning option available is "Partition by field: CustomerID". Do I need to do anything with partitioning?



Great questions! Let's tackle them one at a time:

1. Defining the Primary Key in BigQuery

BigQuery does not inherently support primary keys as traditional relational databases do, because BigQuery is designed to handle large-scale data warehousing operations where the emphasis is on data analysis rather than transactional integrity. However, you can enforce uniqueness and primary key constraints in your application logic or through the use of SQL queries that check for duplicate IDs before inserting new records.

If you require a system to ensure that `CustomerID` is unique, you can do the following:

- **Unique Constraint:** While BigQuery does not natively enforce uniqueness, you can periodically run a deduplication query to ensure no duplicate `CustomerID` exists. Alternatively, when loading data, ensure your data preparation or ETL process includes a step to check for and remove duplicates.
- **Data Insertion Checks:** When inserting data, you can use a query that checks for the existence of the `CustomerID` before inserting a new record.

Here's an example query to check for an existing `CustomerID` before inserting:

sql

```
INSERT INTO your dataset.Customers (CustomerID, CustomerName)
SELECT 1 AS CustomerID, 'New Customer' AS CustomerName
WHERE NOT EXISTS (
  SELECT 1 FROM your dataset.Customers WHERE CustomerID = 1
);
```

This SQL snippet prevents the insertion of a new record with `CustomerID = 1` if such a record already exists.

2. Partitioning and Clustering in BigQuery

Partitioning is used in BigQuery to divide a large table into manageable parts based on the values in one column, typically a date or timestamp, but it can be any field such as `CustomerID`. Partitioning can make queries more efficient by reducing the amount of data scanned.

When to Use Partitioning:

- If you have a very large number of customers and queries frequently involve filtering or aggregating data for specific `CustomerIDs`, then partitioning by `CustomerID` might be useful.
- However, partitioning on `CustomerID` could be less efficient if the number of unique customers is very high, leading to many small partitions. It's generally more common to partition by a date field, such as a creation date, if applicable.

Clustering can be used in conjunction with or independently of partitioning. Clustering sorts data within each partition based on the values in one or more columns. This can significantly improve performance for queries that filter or sort on the clustered columns.

Recommendation:

- **For a customers table:** Unless `CustomerID` covers a range of frequently queried and sufficiently large segments of data, you might not benefit much from partitioning by this field. Consider partitioning by a date-related field if applicable.
- **Use Clustering:** If you decide against partitioning by `CustomerID`, consider clustering the table by `CustomerID` and possibly other columns that are frequently used in your queries to improve performance.

In summary, carefully consider your typical query patterns and data volume when deciding on partitioning and clustering settings. These features can greatly enhance query performance but need to be matched to your specific use case.

It sounds like for the Customers table the best approach would be to not use any partitioning, is that a fair assumption?



Yes, that's a fair assumption. For the `Customers` table, especially if it's not expected to be extremely large (i.e., not containing millions or tens of millions of entries), partitioning may not provide significant benefits and could even complicate your query setup without much payoff.

Here are a few reasons why skipping partitioning might be the best approach for the Customers table:

1. **Simplicity:** Keeping the table structure simple without partitioning means you won't need to manage any additional complexity that comes with partitioned tables, such as dealing with partition filters in your queries.
2. **Size Considerations:** Partitioning is most beneficial for large datasets where queries can benefit from scanning smaller subsets of data. If the `Customers` table is relatively small, the performance gains from partitioning might be negligible.
3. **Query Flexibility:** Without partitioning, you won't need to consider partitioning keys in your queries, which gives you more flexibility in how you write and optimize your SQL.
4. **Cost:** While partitioning can reduce costs by reducing the amount of data scanned during queries, if the overall data volume is low, the cost savings might be minimal.

Clustering, on the other hand, might still be beneficial even if you decide against partitioning. Clustering can organize data based on the `CustomerID` or other fields that are often used in query filters, potentially improving query performance by reducing the amount of data read during query execution.

In conclusion, for the `Customers` table, it's reasonable to opt out of partitioning, especially if the data volume isn't massive. This approach keeps your BigQuery management straightforward while still maintaining good performance for typical usage patterns.

Okay, I've set up all of my BigQuery database tables. Here's a summary of those tables: `Customers`: - `CustomerID` (Integer, Required) - `CustomerName` (String, 255 max length, Required) `Competitors`: - `CompetitorID` (Integer, Required) - `CompetitorName` (String, 255 max length, Required) - `ReviewCount` (Integer, Nullable) - `AverageRateing` (Float, Nullable) `CustomersCompetitors`: - `CustomerID` (Integer, Required) - `CompetitorID` (Integer, Required) `CustomerQueries`: - `QueryID` (Integer, Required) - `Query` (String, 255 max length, Required) - `CustomerID` (Integer, Required)



Great, your database structure looks well-organized and well-suited for the purposes you've described. Here are a few suggestions to ensure you make the most of this setup:

1. **Indexing:** Make sure to index the `CustomerID` and `CompetitorID` fields in the relevant tables to improve the performance of your queries, especially those involving joins across these tables.

2. **Data Integrity:** Consider setting up foreign key-like constraints in your application logic or data loading scripts since BigQuery does not enforce foreign key constraints. This can help maintain referential integrity between tables.
3. **Query Optimization:** When you start running queries, particularly those involving joins across multiple tables, monitor their performance. Utilize BigQuery's query execution details to understand if there are any bottlenecks or optimizations you can make, such as adjusting how you're clustering your data.
4. **Security and Access:** Manage access to your datasets and tables appropriately, ensuring that only authorized users and applications can view or manipulate the data.
5. **Maintenance and Monitoring:** Regularly review the setup to ensure it continues to meet your needs as data volumes grow or usage patterns change. BigQuery offers tools to analyze and optimize storage and query costs.

With these tables configured, you're well-positioned to start integrating and analyzing data from your applications and external APIs. If you need any help with specific queries or optimization tips as you start to work with this data, feel free to ask!

How do we set up the indexing for CustomerID and CompetitorID in BigQuery?



In BigQuery, traditional indexing like you might use in other databases isn't applicable because BigQuery is a fully-managed data warehouse that handles data storage differently to optimize for analytics at scale. Instead of using indexes, BigQuery automatically manages the data storage and retrieval processes to optimize performance.

However, there are ways to optimize your data structure in BigQuery to improve query performance, akin to indexing:

1. Partitioning and Clustering

While BigQuery does not use traditional indexes, you can optimize query performance through partitioning and clustering:

- **Partitioning:** As mentioned previously, partitioning can help reduce the amount of data scanned in queries, which reduces costs and increases query speed. You might partition data on a date column if applicable.

- **Clustering:** You can cluster your tables by one or more columns. For tables like `CustomersCompetitors`, you might cluster by both `CustomerID` and `CompetitorID`. Clustering sorts the data based on the cluster columns within each partition (if the table is partitioned) or within the table if not partitioned.

Here's how you might set up a table with clustering on `CustomerID` and `CompetitorID`:

```
sql
```

```
CREATE TABLE your_dataset.CustomersCompetitors (  
    CustomerID INT64,  
    CompetitorID INT64  
)  
CLUSTER BY CustomerID, CompetitorID;
```

2. Use of Materialized Views

For frequently executed and computationally expensive queries, consider using materialized views. A materialized view is a precomputed view that periodically caches the result of a query for faster access:

This can be especially beneficial for complex joins or aggregations that are frequently used. It's like creating a pre-indexed snapshot of your data based on specific query patterns.

3. Optimizing Query Performance

- **Avoid `SELECT *`:** Always try to specify which columns you need in your `SELECT` statements to avoid scanning unnecessary data.
- **Use `WHERE` clauses effectively:** Make sure your queries include conditions to filter data as much as possible, reducing the amount of data processed.

4. Monitoring and Fine-Tuning

- **Query Execution Details:** BigQuery provides detailed execution plans for your queries, which can help identify which parts of your query are taking the most time or processing the most data.

- **BigQuery Query Plan Explanation:** Use the **EXPLAIN** statement to see a detailed breakdown of how BigQuery plans to execute your SQL queries.

By understanding and utilizing these features, you can significantly optimize how your data is stored and accessed in BigQuery, mimicking the benefits of traditional database indexing and improving your query performance.

We are missing one key aspect of this implementation, which is to store historical data about competitors. We will want to be able to look at the executions of our script and the data it collected for a customer or competitor at various dates. This way we can have a historical record of what our customer data and their competitor data looks like. I've mapped out a new table in our database diagram called "Business Data" table. The Business Data table will store information about the customer or competitor using a BusinessID which can either be the CustomerID or CompetitorID. Can you tell me if this is an adequate setup for our needs?



Your updated diagram and the addition of the "Business Data" table to store historical data about customers and competitors look promising for achieving your goal of tracking changes over time. Here are a few considerations and potential adjustments to ensure your setup is optimized for your needs:

Review of the "Business Data" Table Setup

- **BusinessID:** It's a good idea to use a generic identifier like **BusinessID**, which can reference either a **CustomerID** or a **CompetitorID**. This approach simplifies the schema and makes it versatile to store data for both entities.
- **Query ID:** Including the **Query ID** allows for tracing back which specific queries generated this data, enhancing traceability.
- **Review Count and Velocity:** Storing these metrics with dates allows for trend analysis over time.
- **Date Field:** The inclusion of a **Date** field is crucial as it will enable you to perform time-based queries and see how data trends over different periods.

Recommendations and Considerations

1. **Clarity on BusinessID:** Ensure that there's a clear method to differentiate whether a **BusinessID** refers to a customer or a competitor. You might consider adding an additional column like **BusinessType** with values such as "Customer" or "Competitor" to make queries straightforward and avoid confusion.

2. **Indexing:** Since you'll likely query this table frequently using **BusinessID**, **Query ID**, and **Date**, consider clustering the table based on these columns. This approach will optimize the performance of queries filtering by these attributes.

3. **Data Integrity and Normalization:**

- Make sure there is a robust method to maintain data integrity, especially since **BusinessID** can refer to two different types of entities. Using foreign keys or application-level checks can help maintain consistency.
- If the volume of data becomes significantly large, consider whether partitioning by **Date** might be beneficial for query performance and cost optimization.

4. **Historical Data Management:**

- Determine how often you'll snapshot the data. If you're capturing data changes frequently, ensure that your storage and querying strategy remains cost-effective.
- Consider implementing a data retention policy if historical data grows extensively to manage costs in BigQuery.

5. **Security and Access Control:** Given that this table will accumulate sensitive competitive information over time, carefully manage who has read and write access to this data.

Your planned setup appears well-suited for tracking historical changes in competitor and customer data. Just make sure to address the potential for ambiguity with the **BusinessID** and consider enhancing your table structure for better performance and data management. This proactive planning will help you leverage BigQuery's capabilities efficiently while ensuring your data remains manageable and useful for analytical purposes.

Sorry, I made a mistake in the diagram. The columns for the BusinessData table are as follows: - RecordID (Integer, Required) - BusinessID (Integer, Required) - BusinessType (String, 20 max length, Required) - ReviewCount (Integer, Nullable) - AverageRating (Float, Nullable) - Date (Required) This table was created as described above in BigQuery, and incorporates your suggestion for a BusinessType column which can be "Customer" or "Competitor". I've set the length for the BusinessType to 20 characters in case we expand the functionality of this system to include other options in the future.



Your updated structure for the "BusinessData" table in BigQuery looks well-considered and effectively addresses the requirement to track historical data for both customers and competitors. Here are some key aspects that strengthen your setup:

Key Aspects of the Updated "BusinessData" Table

1. **RecordID:** Using a unique identifier for each record enhances data management and querying efficiency. This field can serve as a primary key within BigQuery, helping to ensure that each entry is unique and easily accessible.
2. **BusinessID and BusinessType:** These fields together provide a flexible yet robust way to distinguish between customers and competitors, as well as potentially other types of business entities in the future. This design choice enhances the adaptability of your database.
3. **ReviewCount and AverageRating:** Storing these metrics allows for detailed analysis over time, providing insights into trends and changes in the review landscape.
4. **Date:** Including a specific date for each record is crucial for historical analysis and trend monitoring. This allows you to perform time-series analyses and track the evolution of data over time.

Additional Recommendations

- **Clustering and Partitioning:** Given that you'll likely query this table frequently by date, consider partitioning the table by the `Date` field. Additionally, clustering the table by `BusinessID` and `BusinessType` can further optimize query performance by organizing the data in a way that aligns with common query patterns.
- **Data Integrity Checks:** Implement checks either within BigQuery (using SQL queries to check for duplicates or inconsistencies before inserting data) or in your application logic to ensure that the data remains consistent and reliable.
- **Data Management Strategies:** As your data grows, consider implementing lifecycle management policies in BigQuery. These can help manage costs and performance by automatically managing older data, such as downgrading the storage class of data that is no longer accessed frequently.
- **Security and Compliance:** Ensure that access controls are appropriately configured to protect sensitive data, especially given that this table will store competitive business information.

Your setup is well-positioned to provide robust historical insights into your customers and competitors, facilitating detailed analytics and strategic decision-making based on historical data trends. If you need further assistance with query optimization or data management strategies in BigQuery, feel free to ask!

BigQuery only lets us use one field to partition the table by, so I am going to set up partitioning on the BusinessID field. When I select the BusinessID field for partitioning, BigQuery is asking for a WHERE clause. The fields required are: Start, End, Interval, and Clustering Order. What should I enter here?



The partition settings you're seeing are specific to range-based partitioning in BigQuery, which allows you to divide a table into segments based on integer values in a specified column. Here's how to configure it based on the BusinessID, and an explanation of each setting:

Range Partitioning Setup

1. **Start:** This is the starting value of your partitioning key range. You'll want to choose a value that reflects the smallest possible **BusinessID** you expect to use. If **BusinessID** starts at 1, then you would set this to 1.
2. **End:** This is the ending value of your partitioning key range and must be greater than the Start value. You need to estimate the highest **BusinessID** you expect to reach. For example, if you don't expect to exceed 10,000 records soon, you might set this to 10,001 (since End is exclusive).
3. **Interval:** This setting defines the range width for each partition. For example, setting this to 100 means that each partition will contain 100 Business IDs. The appropriate value depends on the distribution and total number of Business IDs you expect. If you have thousands of Business IDs, larger intervals might make sense to reduce the number of partitions created.
4. **Clustering Order:** Since BigQuery supports both partitioning and clustering, you can specify one or more columns for clustering within each partition. For the **BusinessData** table, you might consider clustering by **Date** and **BusinessType** to further improve query performance.

Considerations and Recommendation

- **Range Partitioning on BusinessID:** It may not be the best choice if **BusinessID** is not sequentially assigned or if the range of **BusinessID** is too broad, as it can lead to unbalanced partitions or inefficient querying.
- **Alternative Partitioning Approach:** Given that your data has a **Date** field, time-based (date or timestamp) partitioning might be more beneficial, especially for analytical queries that often span time periods. Date-based partitioning would naturally align with querying patterns that analyze trends over time.
- **Using Clustering:** If you opt for date-based partitioning, clustering by **BusinessID** and **BusinessType** would help optimize performance for queries that focus on specific businesses or types of business.

Given these points, if you still wish to proceed with range partitioning by **BusinessID**, the values you choose for Start, End, and Interval should reflect your expected distribution of Business IDs. If your use case strongly supports it, setting it up with reasonable estimates for these values will work; otherwise, reconsidering the partitioning strategy to align with query patterns may be beneficial.

Can partitioning be set up after the table is created, or only at the time of creation?



In BigQuery, partitioning a table needs to be specified at the time of table creation. Once a table is created without partitioning, you cannot directly modify its schema to add partitioning. However, if you need to change the partitioning on an existing table, you can follow these steps:

How to Add Partitioning to an Existing Table

1. Create a New Table with Partitioning:

- Define the schema and partitioning settings when creating a new table.
- You can use the BigQuery web UI, command-line tool, or API to create the table with the desired partition settings.

2. Copy Data from the Old Table:

You can copy the data from the existing non-partitioned table to the new partitioned table. This can be done with a simple SQL query like:

sql

```
CREATE TABLE new partitioned table
PARTITION BY DATE(timestamp column)
AS
SELECT * FROM existing_table;
```

3. Verify the Data:

Ensure that all data has been transferred correctly to the new partitioned table.

4. Delete the Old Table (Optional):

Once you confirm the data is correctly copied and the new table is functioning as expected, you can delete the old table. However, it's often wise to keep the old table until you are completely sure that everything is working as expected.

5. Redirect Queries to the New Table:

Update any queries, views, or data sources that referenced the old table to now use the new partitioned table.

Considerations

- **Downtime:** Depending on the size of the data and the complexity of the schema, this process might take some time and could potentially lead to temporary unavailability of the data.
- **Cost:** Copying data between tables incurs query costs based on the amount of data processed, especially if you are performing a full table scan.
- **Data Consistency:** Ensure that any ongoing data operations that might affect the old table are paused or handled appropriately during the migration to avoid data inconsistencies.

Setting up partitioning correctly from the beginning is crucial because altering partition settings post-creation can be cumbersome and costly. If you anticipate the need for partitioning in the future, it is best to implement it during the initial table setup.

We are expecting to have a maximum of 500 Customer records. For each customer record, we are expecting a maximum of 50 competitors. We are going to be running this system once a month, and storing data for competitors and customers once every month. What values would you recommend that I use in the WHERE clause for the partitioning for BusinessID in the BusinessData table?



Given your specific use case details, setting up partitioning effectively requires a strategic approach to ensure performance optimization and cost efficiency. Considering that each customer could have up to 50 competitors, and you're expecting to update this data monthly, the distribution of **BusinessID** values needs to be analyzed for the most efficient partitioning strategy.

Understanding Your Data Distribution

Maximum Records: If you have up to 500 customers and each could have up to 50 competitors, you potentially have $500 \times 50 = 25,000$ competitor records plus 500 customer records, totaling up to 25,500 potential unique **BusinessID** values if every competitor and customer is assigned a unique ID.

Setting Up Partitioning by BusinessID

Partitioning by BusinessID: Since you are dealing with a defined range of IDs that can be large, range partitioning could be used effectively if your queries frequently involve specific ranges of **BusinessID**. However, the usefulness of this partitioning strategy depends heavily on your typical query patterns.

Recommended Values for WHERE Clause in Range Partitioning

Here's how to configure the range partition settings based on the expected distribution of **BusinessID**:

1. **Start Value:** Begin with the smallest **BusinessID** you expect, which is typically 1.
2. **End Value:** Consider a slightly higher value than your maximum expected **BusinessID** to allow for some growth. If you're expecting around 25,500 records initially, setting this to about 30,000 might be a safe buffer.

3. **Interval:** The interval should be set based on how you expect **BusinessID** values to be accessed. If access patterns are fairly evenly distributed across the range, you could choose an interval that creates a manageable number of partitions. For example, setting an interval of 1,000 might result in about 30 partitions, which could be a good balance between too many small partitions and too few large ones.

Example Configuration

- **Start:** 1
- **End:** 30,000
- **Interval:** 1,000

Alternative Consideration: Partitioning by Date

Given that you are updating this data monthly, another effective strategy might be to partition the table by a **Date** field instead, if your primary query pattern involves filtering by specific months or date ranges. This approach could naturally align with your data update frequency and might be more efficient if your analysis is time-based.

Conclusion

Setting up partitioning effectively requires a deep understanding of both your data and your query needs. If your queries often involve specific ranges of **BusinessID**, the suggested range partitioning could work well. Otherwise, consider whether partitioning by date or even clustering by **BusinessID** within date partitions might be more aligned with how you'll access the data.

Is there a downside to setting this up for growth of up to 2x? So 51,000 total records?



Setting up your partitioning to accommodate up to 2x growth, reaching 51,000 total records, is a proactive approach that can help future-proof your database schema in BigQuery. Here are some considerations and potential downsides to keep in mind when planning for such growth:

Benefits of Planning for Growth

1. **Scalability:** Preparing for a higher range of **BusinessID** values ensures that you won't need to perform costly schema migrations or adjustments as your data grows.
2. **Flexibility:** Allows more room for adding new customers and competitors without immediate concerns about exceeding the partition range limits.

Potential Downsides and Considerations

1. **Partition Efficiency:** The main downside of setting a very wide range for partitioning is the potential inefficiency in partition usage. If the data is not evenly distributed across the entire range, some partitions may end up underutilized, leading to less efficient queries. This is particularly relevant if many **BusinessIDs** cluster in specific ranges and leave other ranges sparse.
2. **Cost Implications:** BigQuery charges for queries based on the amount of data processed. If partitions are not optimally utilized (e.g., many empty or sparse partitions), you may end up scanning more data than necessary, leading to higher query costs. However, this is more of a concern with extremely large datasets or very high query volumes.
3. **Management Complexity:** With a wider range of partitions, managing and monitoring these partitions might be slightly more complex, especially if you need to adjust your setup as your understanding of the data distribution evolves.

Recommendations

Given the scale you are working with (up to 51,000 records expected in the foreseeable future), the potential downsides are likely minimal and manageable, especially when weighed against the benefits of not having to reconfigure your partitioning setup as you grow. Here are some additional tips:

- **Monitor Data Distribution:** Regularly review how your data is distributed across partitions. BigQuery provides detailed information about table storage and usage, which can help you adjust your partitioning strategy if necessary.
- **Consider Clustering:** Alongside range partitioning, consider clustering your table by other key columns like **Date** or **BusinessType** to further enhance query performance. Clustering can help ensure that data within each partition is organized efficiently.
- **Periodic Reviews:** As your database grows and your business evolves, periodically review your partitioning strategy to ensure it still aligns with your data access patterns and performance objectives.

Planning for a 2x growth in your partitioning strategy is generally a sound decision, especially in a managed environment like BigQuery where the management overhead is relatively low compared to traditional database systems.

Can you provide the values I should use for partitioning given a total of 51,000 records?



For partitioning your table in BigQuery to efficiently manage up to 51,000 records, you'll need to set the range partitioning parameters appropriately. Here's a recommendation for setting up the range partitioning with 51,000 records:

Range Partitioning Setup

1. Start Value:

Start from the lowest possible **BusinessID**, which is typically 1, assuming **BusinessID** starts at this value.

2. End Value:

Since the end value in BigQuery's range partitioning is exclusive, you should set this slightly above your maximum expected **BusinessID**. If you're planning for up to 51,000 records, setting the end value at 52,000 or 55,000 provides a buffer that accommodates slight growth beyond your current projection.

3. Interval:

- The interval determines the range width of each partition. To optimize partitioning and balance the number of partitions against the potential query performance benefits, you can choose an interval that results in a manageable number of partitions.
- A common approach is to set the interval so that each partition covers a range of values that is neither too small (leading to too many partitions) nor too large (which could reduce the performance benefits of partitioning).
- For example, setting the interval at 1,000 would result in about 51 or 55 partitions (depending on whether you choose 52,000 or 55,000 as the end value). This should provide a good balance by creating partitions that are not too granular but still effective for query performance.

Example Configuration

- **Start:** 1
- **End:** 55,000
- **Interval:** 1,000

This setup will create partitions that each cover a range of 1,000 unique **BusinessID** values, making it easier for BigQuery to manage and query the data efficiently. This configuration is beneficial if queries frequently filter or perform operations based on ranges of **BusinessID**.

Final Considerations

- **Query Patterns:** Ensure that this partitioning aligns with your common query patterns. For example, if queries frequently access data based on **BusinessID**, this setup is suitable.
- **Monitoring and Adjustment:** After implementing partitioning, monitor the performance and storage metrics provided by BigQuery to see if the partitioning strategy is effective. Be prepared to adjust the parameters if you find that the data distribution or query performance could be optimized further.

By setting up your partitioning parameters in this way, you should achieve a good balance between manageability and performance for your expected scale.

For clustering order, it seems to me that the Date field would make the most sense. Is that a fair conclusion?



Yes, your conclusion to use the **Date** field for clustering in BigQuery is indeed a fair and strategic choice, especially given the nature of your dataset and the likely query patterns. Clustering by **Date** offers several advantages:

Advantages of Clustering by Date

1. **Efficiency in Time-based Queries:** Clustering by **Date** can significantly enhance the performance of queries that filter or aggregate data based on specific time periods. This is particularly useful for historical analyses or trend monitoring, where queries often involve date ranges.
2. **Reduced Costs:** By clustering on the **Date** field, BigQuery can more efficiently locate and process only the relevant partitions of data needed for a query, reducing the amount of data scanned and consequently lowering query costs.
3. **Improved Query Performance:** Clustering organizes the data within each partition into sorted blocks based on the **Date** values. This sorting allows BigQuery to quickly prune large volumes of data that fall outside the time range specified in queries, speeding up the execution time.

Additional Clustering Considerations

- **Multiple Clustering Columns:** While clustering by **Date** is effective, you can also consider clustering by additional dimensions if your queries frequently involve other specific fields. For instance, clustering by both **Date** and **BusinessType** might be beneficial if you often need to perform operations that span both these attributes.

- **Alignment with Partitioning:** If you have partitioned the data (for example, by **BusinessID** or another field), adding clustering on **Date** complements the partitioning by further refining the organization of data within each partition, enhancing the query performance even more.

Implementing Clustering

When you create or modify a table to include clustering in BigQuery, specify the clustering columns in the table creation statement. For your case, here's how you might specify clustering when creating a table:

sql

```
CREATE TABLE your_dataset.your_table_name (  
  RecordID INT64,  
  BusinessID INT64,  
  BusinessType STRING,  
  ReviewCount INT64,  
  AverageRating FLOAT64,  
  Date DATE  
)  
PARTITION BY DATE  
CLUSTER BY Date  
OPTIONS(  
  description "A table to store business data with partitioning by Date and  
  clustering by Date for optimal query performance."  
);
```

This SQL statement sets up a table with **Date** as both the partitioning and clustering column. Adjust the **PARTITION BY** clause if you decide to partition by a different field.

In conclusion, clustering by **Date** in your scenario is an excellent strategy for optimizing data access patterns focused on time-based queries, which are common in analytical and historical data analysis environments. This setup should provide a solid foundation for efficient and cost-effective data management in BigQuery.

Okay, great. Here is the final BusinessData table: - RecordID (Integer, Required) - RecordDate (Date, Required) **Change name to avoid using a reserved word like 'date' - BusinessID (Integer, Required) - BusinessType

(String, 20 max length, Required) - BusinessName (String, 255 max length, Required) - ReviewCount (Integer, Nullable) - AverageRating (Float, Nullable) I added a column for "BusinessName" because although that is stored in the Customers or Competitors tables, businesses sometimes change their Google Business Profile name. Adding the BusinessName column to the BusinessData table will allow us to have a historical record of the business name at the time of the scan.



Your final setup for the **BusinessData** table looks comprehensive and well-suited to your needs. Here are some key points about your final table configuration:

Key Points of Your Final BusinessData Table Setup

1. **RecordID**: Serves as a unique identifier for each record, ensuring easy and accurate data retrieval.
2. **RecordDate**: Renaming the **Date** field to **RecordDate** is a wise choice to avoid confusion with the reserved keyword and improve readability in your queries.
3. **BusinessID**: Continues to serve as a key field to link data across your different tables.
4. **BusinessType**: Useful for differentiating between Customers and Competitors within your data queries, enhancing flexibility.
5. **BusinessName**: Adding the **BusinessName** column is a strategic move. It allows for capturing the business name at the time of data entry, which is crucial for tracking changes over time, especially given that business names can change.
6. **ReviewCount** and **AverageRating**: These fields are essential for tracking the performance metrics of businesses over time, providing insights into trends and changes in performance metrics.

Additional Recommendations

- **Partitioning and Clustering**: Consider partitioning this table by **RecordDate** to optimize query performance over time-based data, and cluster by **BusinessID** and **BusinessType** to further enhance data retrieval efficiency based on your frequent query patterns.
- **Data Integrity and Validation**: Ensure that data entered into this table is validated, especially for the non-required fields like **ReviewCount** and **AverageRating**, to maintain data quality.

- **Security and Access Control:** Set appropriate access controls to protect sensitive business data, particularly because this table contains detailed and potentially sensitive business information.

This table structure should serve you well in managing and analyzing business data over time, providing valuable insights into changes and trends in business metrics and names. If you need further assistance with specific queries or additional configuration advice, feel free to ask!

Okay great. Now, in preparation for writing the code for the Google Cloud Function we'll create, can you tell me what I need to know about allowing the Cloud Function script to connect to BigQuery? What information will be required to connect to the BigQuery database? What steps (if any) do I need to take in BigQuery to allow for that connection?



Connecting a Google Cloud Function to BigQuery involves several steps to ensure secure and efficient access. Here's what you need to know and do:

Prerequisites

1. **Google Cloud Project:** Ensure your Cloud Function and BigQuery are in the same Google Cloud Project, or that you have set up appropriate permissions across projects.
2. **Google Cloud SDK:** Typically, Cloud Functions use the Google Cloud SDK to interact with BigQuery. This SDK is pre-installed in the Cloud Functions environment.
3. **IAM Permissions:** The executing identity (service account) of the Cloud Function needs appropriate permissions to access BigQuery.

Configuring IAM Permissions

1. **Service Account:** By default, a Cloud Function uses the default App Engine service account (`PROJECT_ID@appspot.gserviceaccount.com`), but you can create and use a custom service account if needed for finer control.

2. Granting Permissions:

- Ensure the service account used by your Cloud Function has at least **BigQuery Data Editor** and **BigQuery Job User** roles. These permissions allow the function to run queries and manage data within BigQuery.
- You can assign these roles in the Google Cloud Console under IAM & Admin.

Steps to Connect to BigQuery in Cloud Function

1. **Using the Google Client Libraries:** Google provides client libraries for interacting with BigQuery. For Node.js, for example, you would use the `@google-cloud/bigquery` library. Here's a basic setup:

```
javascript

const {BigQuery} = require('@google-cloud/bigquery');
const bigquery = new BigQuery();

async function queryBigQuery() {
  const query = `SELECT * FROM \`${your_dataset.your_table}\``;
  const options = {
    query: query,
  };

  const [rows] = await bigquery.query(options);
  console.log('Query Results:', rows);
}
```

2. **Environment Variables:** Optionally, use environment variables to manage configurations like dataset names or table names, which helps keep your code flexible and secure.
3. **Error Handling:** Implement robust error handling and logging within your function to manage and troubleshoot any issues with BigQuery connectivity or query execution.
4. **Testing:** Before deploying, test your function locally or in a development environment to ensure it properly connects and performs operations in BigQuery.

Security Considerations

- **Least Privilege:** Grant only the necessary permissions to the service account to minimize security risks.
- **Secure Coding Practices:** Ensure that your function's code securely handles data and errors to avoid exposing sensitive information.

Deployment

Deploy your function via the Google Cloud Console, gcloud command line, or through CI/CD pipelines, ensuring you've set the correct trigger (HTTP, Pub/Sub, etc.) and region.

By following these steps, your Google Cloud Function will be able to securely connect to and interact with BigQuery, performing data manipulations and retrievals as needed. If you need specific details for another programming language or further configuration specifics, let me know!

Great, I've created a Cloud Function with the name "GbpReviewsMonitoringScrape". The Authentication setting for the function is set to "Require Authentication: Manage authorized users with Cloud IAM". The trigger type is HTTP. The URL for the Cloud Function is:

[REDACTED]



Great, it sounds like you have your Cloud Function set up and ready to go! With the authentication set to "Require Authentication" and managed via Cloud IAM, you ensure that only authorized users and services can invoke your function, adding an important layer of security.

Next Steps

Here are a few recommendations and next steps to consider with your new Cloud Function:

1. Testing the Function:

- **Manual Testing:** You can manually test the function by sending HTTP requests to the function's URL using tools like Postman or cURL. This is a quick way to ensure that it's responding as expected.
- **Automated Testing:** Consider setting up automated tests that can be run as part of your deployment process to ensure ongoing reliability.

2. Monitoring and Logging:

- **Google Cloud Monitoring:** Use Google Cloud Monitoring to set up alerts based on the function's execution metrics and logs.
- **Google Cloud Logging:** Ensure that logging is properly configured in your function to capture both operational logs and application-specific events. This will be crucial for diagnosing issues and understanding the function's operational health.

3. Access Control:

- **Review IAM Settings:** Make sure that the IAM settings are correctly configured to allow only the necessary entities to invoke your function.
- **Service Account Permissions:** Double-check that the service account associated with your Cloud Function has the appropriate permissions for all required actions, especially interactions with BigQuery.

4. Optimizing Cloud Function Performance:

- **Memory and Timeout Settings:** Adjust the memory allocation and timeout settings based on the function's performance and needs.
- **Cold Starts:** If the function's start-up time is a concern (commonly referred to as "cold starts"), consider strategies to mitigate this, such as keeping the function warm with scheduled invocations if appropriate.

5. Documentation:

Function Documentation: Document the purpose, usage, and any relevant details about the function. This documentation will be valuable for both current understanding and future maintenance.

6. Integrate with Other Services:

If your function needs to interact with other services or triggers within your Google Cloud environment (like from a Pub/Sub topic or another Cloud Function), set up those integrations and ensure they are securely authenticated.

By following these steps, you'll not only ensure that your Cloud Function is secure and reliable but also well-integrated and monitored within your broader cloud architecture. If you have any specific questions about the function's implementation or encounter any issues during testing, feel free to ask for help!

I have set the Cloud Function to use the Python 3.10 runtime. I set the entry point for the function to "script_init". Please write the python script to connect to the BigQuery database we created. As a test for now, please query the Customers table for all records and output them in a log or other debugging method.



To write a Python script for your Google Cloud Function that connects to BigQuery and queries the Customers table, you'll need to use the `google-cloud-bigquery` client library. Below is a sample script that sets up the Cloud Function to fetch all records from the `Customers` table and log the output.

Pre-requisites:

Make sure the `google-cloud-bigquery` library is included in your `requirements.txt` file:

```
google-cloud-bigquery==2.34.0
```

Python Script for the Cloud Function:

Here's a complete Python script for the function named `script_init`:

python

```
from google.cloud import bigquery
import json

def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
        <http://flask.pocoo.org/docs/1.0/api/#flask.Request>
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`
        <http://flask.pocoo.org/docs/1.0/api/#flask.Flask.make_response>.
    """
    # Instantiate a BigQuery client
    client = bigquery.Client()

    # Define the query
    query = """
        SELECT * FROM `your-project-id.your-dataset.Customers`
    """

    # Perform the query
    query_job = client.query(query) # Make an API request

    # Collect and log the results
    customers = query_job.result() # Waits for the job to complete

    # Output the results
    for customer in customers:
        print(json.dumps({"CustomerID": customer.get("CustomerID"),
            "CustomerName": customer.get("CustomerName")}))

    return "Query executed successfully!"
```

Notes on the Script:

1. **Client Initialization:** The script initializes a BigQuery client using the `google.cloud.bigquery` library. This client handles all interactions with the BigQuery service.

2. **Query Execution:** The script defines and executes a SQL query to fetch all records from the Customers table.
3. **Result Handling:** After executing the query, the script iterates through the results and prints them. In a real-world scenario, you might want to process these results differently.
4. **Permissions:** Ensure that the service account used by the Cloud Function has appropriate permissions in BigQuery to execute queries and access the dataset and tables.

Deployment and Testing:

- Deploy this script to your Cloud Function.
- Invoke the function using the provided HTTPS URL and observe the logs for output.

This script should meet your requirements for connecting to BigQuery and fetching data from the Customers table. Modify the project ID, dataset, and table names as necessary to match your actual BigQuery setup. If you encounter any issues or need further customization, feel free to ask!

The following line was already present in the requirements.txt file: `functions-framework==3.*` The main.py sample code also included the following line at the beginning of the script code: `import functions_framework` Do we need to keep these lines of code?



Yes, you should keep the `functions-framework` line in your `requirements.txt` file and the `import functions_framework` line in your script if you plan to use the Functions Framework for local development and testing of your Cloud Function. Here's why each is important:

Functions Framework

1. Functions Framework:

- The Functions Framework is an open-source project that lets you run your Google Cloud Functions locally and test them before deploying to the cloud. It helps simulate the Cloud Functions environment on your local machine.
- It allows for running, debugging, and testing HTTP-triggered functions with ease.

2. Using Functions Framework:

- If you plan to test your function locally, you would use this framework to start a local server that mimics the cloud environment.
- When you deploy the function to Google Cloud, the framework isn't explicitly needed in the cloud environment, but it's useful during development and testing stages.

Sample Code Adjustment

Since you are using the Functions Framework, you should define your function with the framework's decorators or setup. Here's how you can adjust the sample code using the Functions Framework:

python

```
import functions_framework
from google.cloud import bigquery
import json

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    # Instantiate a BigQuery client
    client = bigquery.Client()

    # Define the query
    query = """
        SELECT * FROM `your-project-id.your-dataset.Customers`
    """

    # Perform the query
    query_job = client.query(query) # Make an API request

    # Collect and log the results
    customers = query_job.result() # Waits for the job to complete

    # Output the results
    output = []
    for customer in customers:
        output.append({
            "CustomerID": customer.get("CustomerID"),
            "CustomerName": customer.get("CustomerName")
        })

    # Return the results as JSON
    return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}
```

Explanation

- **Decorator:** The `@functions_framework.http` decorator is used to define your function as an HTTP-triggered function when running locally. This decorator is compatible with the Google Cloud Functions deployment as well.
- **Response:** The function now returns a JSON-formatted response with a 200 HTTP status code and sets the `Content-Type` to `application/json`, making it more suitable for returning data to an HTTP client.

Testing Locally

To run your function locally using the Functions Framework, you would use a command like this:

```
bash
```

```
functions-framework --target=script_init --port=8080
```

This command starts a local server at `http://localhost:8080`, and you can test your function by sending HTTP requests to this endpoint.

Conclusion

Keeping the `functions-framework` package allows you to simulate and test your Cloud Function locally before deploying it to Google Cloud, providing a smooth development workflow.

In the following line of code, how do I find the values to use for "your-project-id" and "your-dataset"? `query = """ SELECT * FROM your-project-id.your-dataset.Customers """`



To determine the values for `your-project-id` and `your-dataset`, you'll need to reference your Google Cloud Project and BigQuery setup. Here's how you can find each of these values:

Finding Your Project ID

1. Google Cloud Console:

- Navigate to the [Google Cloud Console](#).
- In the top navigation bar, you will see a dropdown for "Project". The Project ID is typically shown alongside the project name. It usually looks something like `my-gcp-project-123456`.

2. Billing and Settings:

Alternatively, you can go to **IAM & Admin > Settings** in the Cloud Console to view your project information, including the Project ID.

Finding Your Dataset Name

1. BigQuery Console:

- Go to the [BigQuery Console](#).
- In the Explorer panel on the left, expand your project to view the datasets.
- The dataset you created should be listed there. For example, if you named your dataset `CustomerGbpCompetitors`, that would be the value to use.

Putting It All Together

Once you have both values, replace `your-project-id` and `your-dataset` in your SQL query string with these actual values. Here's an example:

```
python
```

```
query = """
    SELECT * FROM `my-gcp-project-123456.CustomerGbpCompetitors.Customers`
    """
```

Considerations

- **Project ID:** Ensure you are using the correct project ID associated with the resources you wish to access.
- **Dataset Name:** Verify the dataset name is spelled correctly and matches what you see in the BigQuery Console.

Testing

After updating the query with your actual values, test your Cloud Function locally or deploy it and invoke it to ensure it can successfully connect to BigQuery and retrieve the expected data.

By following these steps, you'll have the correct identifiers needed to access your BigQuery dataset and tables from your Cloud Function.

Okay, here is my Project ID: `agency-seo-tools-318618`



Great! Now that you have your Project ID (`agency-seo-tools-318618`), you'll just need to ensure you have the correct dataset name. Based on our previous discussions, it seems your dataset might be named `CustomerGbpCompetitors`.

Assuming this is correct, your SQL query in the Cloud Function should look like this:

```
python
```

```
query = """
    SELECT * FROM `agency-seo-tools-
318618.CustomerGbpCompetitors.Customers`
    """
```

Verifying the Dataset Name

1. Check in BigQuery Console:

- Go to the [BigQuery Console](#).
- Expand the project `agency-seo-tools-318618` in the Explorer panel.
- Verify that `CustomerGbpCompetitors` is listed as one of the datasets.

Final Script Example

Here's how your script should look with the correct Project ID and dataset:

python

```
import functions_framework
from google.cloud import bigquery
import json

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    # Instantiate a BigQuery client
    client = bigquery.Client()

    # Define the query with your project ID and dataset
    query = """
        SELECT * FROM `agency-seo-tools-
318618.CustomerGbpCompetitors.Customers`
    """

    # Perform the query
    query job    client.query(query) # Make an API request

    # Collect and log the results
    customers    query job.result() # Waits for the job to complete

    # Output the results
    output    []
    for customer in customers:
        output.append({
            "CustomerID": customer.get("CustomerID"),
            "CustomerName": customer.get("CustomerName")
        })

    # Return the results as JSON
    return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}
```

Testing

-
- Deploy this updated function to your Google Cloud Function.
 - Invoke the function using the provided HTTP URL to see if it correctly retrieves and logs the customer data.

If you encounter any issues or need further adjustments, feel free to ask!

Can you write a BigQuery query that inserts the following data into the Customers table: CustomerID | Customer Name 1 | [REDACTED]
[REDACTED] 2 | KCS Kitchener Cleaning Services 3 | [REDACTED]
4 | [REDACTED]



Certainly! To insert data into the `Customers` table in BigQuery, you can use the `INSERT` statement. Here is the SQL query to insert the specified customer data into your `Customers` table:

sql

```
INSERT INTO `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`  
(CustomerID, CustomerName)  
VALUES  
  (1, '[REDACTED]'),  
  (2, 'KCS Kitchener Cleaning Services'),  
  (3, '[REDACTED]'),  
  (4, '[REDACTED]');
```

Explanation

- **Table Reference:** Make sure the table reference `agency-seo-tools-318618.CustomerGbpCompetitors.Customers` is correct, based on your dataset and table name in BigQuery.
- **Columns:** The `INSERT` statement specifies the columns `CustomerID` and `CustomerName`, matching the columns in your table.
- **Values:** Each pair of parentheses contains the values for a new row in the `Customers` table.

Executing the Query

1. BigQuery Console:

- Go to the [BigQuery Console](#).
- Open the query editor and paste the SQL statement above.
- Click "Run" to execute the query and insert the data into your table.

2. Using the BigQuery Client Library:

If you want to execute this query programmatically (e.g., from your Cloud Function or a Python script), you can use the BigQuery client library.

Here's how you might execute this insert operation using Python:

```
python

from google.cloud import bigquery

# Instantiate a BigQuery client
client = bigquery.Client()

# Define the query
query = """
    INSERT INTO `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`
    (CustomerID, CustomerName)
    VALUES
    (1, '████████████████████'),
    (2, 'KCS Kitchener Cleaning Services'),
    (3, '████████████████████'),
    (4, '████████████████████');
    """

# Execute the query
query job  client.query(query)

# Wait for the query to finish
query job.result()

print("Data inserted successfully.")
```

Considerations

- **Data Uniqueness:** Ensure that the `CustomerID` values are unique if your application logic assumes `CustomerID` as a unique identifier.
- **Error Handling:** Implement error handling in your code to manage potential issues during the data insertion process, such as duplicate entries or connection problems.

Feel free to let me know if you need further assistance or if there's anything else you'd like to do with your BigQuery setup!

```
I tried testing the Cloud Function and I received the following error in the
Cloud Shell. Can you tell me if you have any ideas what went wrong?
[5:13:14 PM] - [2024-08-08 21:13:14,991] ERROR in app: Exception on /
[POST] Traceback (most recent call last): File
"/layers/google.python.pip/pip/lib/python3.10/site-packages/flask/app.py",
line 1473, in wsgi_app response = self.full_dispatch_request() File
"/layers/google.python.pip/pip/lib/python3.10/site-packages/flask/app.py",
line 882, in full_dispatch_request rv = self.handle_user_exception(e) File
"/layers/google.python.pip/pip/lib/python3.10/site-packages/flask/app.py",
line 880, in full_dispatch_request rv = self.dispatch_request() File
"/layers/google.python.pip/pip/lib/python3.10/site-packages/flask/app.py",
line 865, in dispatch_request return
self.ensure_sync(self.view_functions[rule.endpoint])(**view_args) # type:
ignore[no-any-return] File
"/layers/google.python.pip/pip/lib/python3.10/site-
packages/functions_framework/execution_id.py", line 106, in wrapper
return view_function(*args, **kwargs) File
"/layers/google.python.pip/pip/lib/python3.10/site-
packages/functions_framework/__init__.py", line 142, in view_func return
function(request._get_current_object()) File
"/layers/google.python.pip/pip/lib/python3.10/site-
packages/functions_framework/__init__.py", line 121, in wrapper return
func(*args, **kwargs) File "/workspace/main.py", line 22, in script_init
query_job = client.query(query) # Make an API request File
"/layers/google.python.pip/pip/lib/python3.10/site-
packages/google/cloud/bigquery/client.py", line 3391, in query future =
do_query() File "/layers/google.python.pip/pip/lib/python3.10/site-
packages/google/cloud/bigquery/client.py", line 3368, in do_query
query_job._begin(retry=retry, timeout=timeout) File
"/layers/google.python.pip/pip/lib/python3.10/site-
packages/google/cloud/bigquery/job/query.py", line 1297, in _begin
super(QueryJob, self)._begin(client=client, retry=retry, timeout=timeout)
File "/layers/google.python.pip/pip/lib/python3.10/site-
```

```
packages/google/cloud/bigquery/job/base.py", line 510, in _begin
api_response = client._call_api( File
"/layers/google.python.pip/pip/lib/python3.10/site-
packages/google/cloud/bigquery/client.py", line 782, in _call_api return
call() File "/layers/google.python.pip/pip/lib/python3.10/site-
packages/google/api_core/retry/retry_unary.py", line 293, in
retry_wrapped_func return retry_target( File
"/layers/google.python.pip/pip/lib/python3.10/site-
packages/google/api_core/retry/retry_unary.py", line 153, in retry_target
_retry_error_helper( File "/layers/google.python.pip/pip/lib/python3.10/site-
packages/google/api_core/retry/retry_base.py", line 212, in
_retry_error_helper raise final_exc from source_exc File
"/layers/google.python.pip/pip/lib/python3.10/site-
packages/google/api_core/retry/retry_unary.py", line 144, in retry_target
result = target() File "/layers/google.python.pip/pip/lib/python3.10/site-
packages/google/cloud/_http/__init__.py", line 494, in api_request raise
exceptions.from_http_response(response)
google.api_core.exceptions.NotFound: 404 POST
https://bigquery.googleapis.com/bigquery/v2/projects//jobs?
prettyPrint=false: Request couldn't be served. Location: None Job ID:
622e13f8-f8a6-44e5-8943-d590a9c5d947
```



The error you're encountering indicates a **404 Not Found** response from the BigQuery API, which suggests that the request to execute a query couldn't be completed. This could be due to several reasons. Let's walk through the most common causes and how to resolve them:

Possible Causes and Solutions

1. Missing or Incorrect Project ID:

- **Cause:** The error message includes a **404** status and mentions that the project ID is missing or incorrect in the API request URL. This typically occurs when the BigQuery client is not properly configured with the project ID.
- **Solution:** Ensure that the project ID is specified correctly when initializing the BigQuery client. You can explicitly set the project ID like this:

```
python
```

```
client = bigquery.Client(project="agency-seo-tools-318618")
```

2. Dataset or Table Not Found:

- **Cause:** The dataset or table specified in your query might not exist in the specified project, or the names might be misspelled.
- **Solution:** Double-check the dataset and table names to ensure they are correct and exist within the specified project. Verify the spelling and the case sensitivity of the names in the BigQuery Console.

3. Service Account Permissions:

- **Cause:** The service account used by your Cloud Function may not have the required permissions to access BigQuery resources.
- **Solution:** Ensure that the service account associated with your Cloud Function has the **BigQuery Data Viewer**, **BigQuery Data Editor**, or **BigQuery User** roles, depending on what actions are needed. You can set these roles in the IAM & Admin section of the Google Cloud Console.

4. Invalid Query Syntax:

- **Cause:** There could be a syntax error in the SQL query itself.
- **Solution:** Review the query for any potential syntax issues. Ensure that backticks (`) are used correctly around project, dataset, and table names, especially if they contain special characters or are reserved words.

Example of a Corrected Code Block

Ensure your BigQuery client is correctly instantiated and the query is accurate:

python

```
import functions_framework
from google.cloud import bigquery
import json

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    # Instantiate a BigQuery client with the correct project ID
    client = bigquery.Client(project="agency-seo-tools-318618")

    # Define the query with your correct project ID and dataset
    query = """
        SELECT * FROM `agency-seo-tools-
318618.CustomerGbpCompetitors.Customers`
        """

    # Perform the query
    try:
        query_job = client.query(query) # Make an API request
        customers = query_job.result() # Waits for the job to complete

        # Output the results
        output = []
        for customer in customers:
            output.append({
                "CustomerID": customer.get("CustomerID"),
                "CustomerName": customer.get("CustomerName")
            })

        # Return the results as JSON
        return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}
    except Exception as e:
        return f"An error occurred: {str(e)}", 500
```

Additional Steps

- **Logging:** Use logging to capture more details in the Cloud Functions logs, which can help identify any other issues.
- **Testing in Console:** Manually verify that you can run the SQL query in the BigQuery Console to ensure it returns results as expected.

By addressing these common issues, you should be able to resolve the error and successfully connect to and query your BigQuery dataset from your Cloud Function. If the issue persists, consider enabling more detailed logging or checking for additional error messages in the Cloud Functions logs.

That worked! Okay, now let's populate some data into the BigQuery database for CustomerQueries so we can test a lookup of all customers and their respective queries. Can you write an SQL query that will add the following queries for each customer: [REDACTED] :

[REDACTED] : [REDACTED] KCS Kitchener
Cleaning Services: commercial cleaning commercial cleaning cambridge
commercial cleaning company house cleaners cambridge house cleaning
cambridge [REDACTED] : [REDACTED]
[REDACTED]



Certainly! To insert the specified queries into the `CustomerQueries` table in BigQuery for each customer, you can use an `INSERT` statement. Here's the SQL query to add the data:

sql

```
INSERT INTO `agency-seo-tools-318618.CustomerGbpCompetitors.CustomerQueries` (QueryID, Query, CustomerID)
VALUES
  [REDACTED]
  (1, '[REDACTED]', 1),
  (2, '[REDACTED]', 1),
  (3, '[REDACTED]', 1),
  (4, '[REDACTED]', 1),
  (5, '[REDACTED]', 1),
  (6, '[REDACTED]', 1),

  [REDACTED]
  (7, '[REDACTED]', 3),
  (8, '[REDACTED]', 3),

  -- KCS Kitchener Cleaning Services
  (9, 'commercial cleaning', 2),
  (10, 'commercial cleaning cambridge', 2),
  (11, 'commercial cleaning company', 2),
  (12, 'house cleaners cambridge', 2),
  (13, 'house cleaning cambridge', 2),

  [REDACTED]
  (14, '[REDACTED]', 4),
  (15, '[REDACTED]', 4),
  (16, '[REDACTED]', 4),
  (17, '[REDACTED]', 4);
```

Explanation

- **QueryID:** This is a unique identifier for each query. Ensure that these IDs are unique within the table and increment sequentially.
- **Query:** The actual search query string associated with each customer.
- **CustomerID:** This refers to the **CustomerID** from the **Customers** table, linking each query to the appropriate customer.

Executing the Query

1. BigQuery Console:

- o Open the [BigQuery Console](#).
- o In the query editor, paste the SQL query above.
- o Click "Run" to execute the query and insert the data into your `CustomerQueries` table.

2. Python Script (Optional):

You can also execute this query programmatically using a Python script with the BigQuery client library if you prefer.

Testing the Insertions

Once the data is inserted, you can test fetching all customers and their respective queries using the following query:

sql

```
SELECT
  c.CustomerName,
  q.Query
FROM
  `agency-seo-tools-318618.CustomerGbpCompetitors.Customers` c
JOIN
  `agency seo tools 318618.CustomerGbpCompetitors.CustomerQueries` q
ON
  c.CustomerID = q.CustomerID
ORDER BY
  c.CustomerName, q.Query;
```

This query joins the `Customers` and `CustomerQueries` tables to list all customers along with their associated queries, ordered alphabetically by customer name and query.

Let me know if you need further assistance or if you'd like to test more scenarios!

Great, now let's update the python script in the Cloud Function to grab customers and queries and output the data as JSON



Sure! Let's update the Python script for your Cloud Function to query both the **Customers** and **CustomerQueries** tables and return the data in a structured JSON format. Here's how you can modify the script to achieve this:

Updated Python Script

python

```
import functions_framework
from google.cloud import bigquery
import json

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    # Instantiate a BigQuery client
    client = bigquery.Client(project="agency-seo-tools-318618")

    # Define the query to join Customers and CustomerQueries
    query = """
        SELECT
            c.CustomerName,
            q.Query
        FROM
            `agency seo tools 318618.CustomerGbpCompetitors.Customers` c
        JOIN
            `agency seo tools
318618.CustomerGbpCompetitors.CustomerQueries` q
        ON
            c.CustomerID = q.CustomerID
        ORDER BY
            c.CustomerName, q.Query;
    """

    # Perform the query
    try:
        query_job = client.query(query) # Make an API request
        results = query_job.result() # Waits for the job to complete

        # Create a dictionary to hold customers and their queries
        customer_queries = {}
        for row in results:
            customer_name = row.get("CustomerName")
            query_text = row.get("Query")

            if customer_name not in customer_queries:
```

```

        customer_queries[customer_name] = []

        customer_queries[customer_name].append(query_text)

    # Convert the dictionary to JSON
    response_json = json.dumps(customer_queries, indent=2)

    # Return the results as JSON
    return response_json, 200, {'Content Type': 'application/json'}

except Exception as e:
    # Return an error message if something goes wrong
    return f"An error occurred: {str(e)}", 500

```

Explanation

- **Join Query:** The SQL query joins the `Customers` and `CustomerQueries` tables to get a list of all customers and their associated queries.
- **Data Structure:** The script organizes the data into a dictionary where each customer name is a key, and the associated queries are stored in a list.
- **JSON Output:** The dictionary is then converted to a JSON string using `json.dumps`, with indentation for readability.
- **Error Handling:** The script includes basic error handling to return an appropriate message if something goes wrong during execution.

Testing

1. **Deploy the Updated Function:** Deploy the updated script to your Cloud Function.
2. **Invoke the Function:** Use the provided HTTP URL to send a request to the function and observe the JSON output.
3. **Review the JSON:** Ensure the JSON output is structured correctly, with each customer and their associated queries listed.

This setup should provide you with a clear JSON representation of your customers and their queries, making it easier to use this data for further processing or display in an application. If you encounter any issues or need additional features, feel free to ask!

This worked perfectly. The DataForSEO API required a location to use for the queries. We need to incorporate a SearchLocation column into our database that will be used to perform the queries. I don't think we'll want to support different locations for each query, so I think the best place for the SearchLocation field would be in the Customers table. What do you think?



Adding a `SearchLocation` column to the `Customers` table is a logical choice for several reasons:

Benefits of Adding SearchLocation to Customers Table

1. **Simplicity:** By associating a single search location with each customer, you simplify the management and execution of queries. This way, each customer has a consistent search location used across all their queries.
2. **Consistency:** It ensures that all queries for a given customer are performed in the same location context, which can lead to more consistent and comparable results over time.
3. **Scalability:** If you later decide to support multiple locations per customer, you could consider expanding the schema or logic to handle this complexity. Starting with a single location simplifies the initial implementation.
4. **Ease of Integration:** Adding `SearchLocation` to the `Customers` table makes it easier to integrate location data into your existing queries without modifying the structure of the `CustomerQueries` table.

Implementation Steps

1. **Update the Customers Table Schema:** Add a `SearchLocation` column to the `Customers` table in BigQuery. This column can store location identifiers or names (e.g., city names, region codes, etc.).
2. **Populate SearchLocation Data:** Update your existing customer records to include appropriate search locations. You can use a mix of manual entry and automated scripts to populate this data based on your needs.
3. **Modify the Cloud Function:** Update your Cloud Function to retrieve the `SearchLocation` for each customer and pass it to the DataForSEO API when performing queries.

SQL to Add SearchLocation Column

Here's how you might alter the `Customers` table to add the `SearchLocation` column:

sql

```
ALTER TABLE `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`  
ADD COLUMN SearchLocation STRING;
```

Example Data Population

After adding the column, you can insert location data like this:

sql

```
UPDATE `agency seo tools 318618.CustomerGbpCompetitors.Customers`  
SET SearchLocation = 'Toronto'  
WHERE CustomerName = '████████████████████';
```

```
UPDATE `agency seo tools 318618.CustomerGbpCompetitors.Customers`  
SET SearchLocation = 'Winnipeg'  
WHERE CustomerName = '████████████████████';
```

```
UPDATE `agency seo tools 318618.CustomerGbpCompetitors.Customers`  
SET SearchLocation = 'Cambridge'  
WHERE CustomerName = 'KCS Kitchener Cleaning Services';
```

```
UPDATE `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`  
SET SearchLocation = 'Toronto'  
WHERE CustomerName = '████████████████████';
```

Updating the Cloud Function

In your Cloud Function, you can now retrieve `SearchLocation` for each customer and incorporate it into your API call logic. Here's a brief update to the query to include `SearchLocation`:

python

```
query = """
SELECT
    c.CustomerName,
    q.Query,
    c.SearchLocation
FROM
    `agency seo tools 318618.CustomerGbpCompetitors.Customers` c
JOIN
    `agency seo tools 318618.CustomerGbpCompetitors.CustomerQueries` q
ON
    c.CustomerID = q.CustomerID
ORDER BY
    c.CustomerName, q.Query;
"""
```

Conclusion

By incorporating `SearchLocation` into the `Customers` table, you maintain simplicity and ensure consistent query execution. This setup should help you efficiently integrate location-based search data into your workflow, enhancing the quality and relevance of your SEO insights. If you need further assistance or additional features, feel free to ask!

I want the `SearchLocation` column to be required. Can you update the SQL? Also for populating the locations in the table, the locations have to use this format: `City,State,Country`. For example, `Cambridge,Ontario,Canada`. Can you update that SQL as well?



Certainly! Here's how you can update the SQL to add the `SearchLocation` column as a required field (not nullable) in the `Customers` table, and how to populate it with location data in the specified format:

Updating the Customers Table Schema

To make the `SearchLocation` column required, you'll specify it as `STRING` and ensure it is non-nullable. Here's the SQL to add the column:

sql

```
ALTER TABLE `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`  
ADD COLUMN SearchLocation STRING NOT NULL;
```

Populating SearchLocation Data

Here's the SQL to update your existing records with the specified location format:

sql

```
UPDATE `agency seo tools 318618.CustomerGbpCompetitors.Customers`  
SET SearchLocation = 'Toronto,Ontario,Canada'  
WHERE CustomerName = '██████████';
```

```
UPDATE `agency seo tools 318618.CustomerGbpCompetitors.Customers`  
SET SearchLocation = 'Winnipeg,Manitoba,Canada'  
WHERE CustomerName = '██████████';
```

```
UPDATE `agency seo tools 318618.CustomerGbpCompetitors.Customers`  
SET SearchLocation = 'Cambridge,Ontario,Canada'  
WHERE CustomerName = 'KCS Kitchener Cleaning Services';
```

```
UPDATE `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`  
SET SearchLocation = 'Toronto,Ontario,Canada'  
WHERE CustomerName = '██████████';
```

Explanation

- **NOT NULL Constraint:** By setting the `SearchLocation` column as `NOT NULL`, you ensure that each customer record must have a valid location specified. This is enforced at the database level, preventing the insertion of records without a search location.
- **Location Format:** The `City, State, Country` format is used to ensure consistency and to meet the API requirements for location-based searches.

Considerations

- **Existing Records:** Before applying the `NOT NULL` constraint, ensure that all existing records in your table have a valid `SearchLocation` value to avoid errors.
- **Data Validation:** Consider implementing data validation checks in your application logic to verify that new records conform to the required location format.
- **Error Handling:** Be prepared to handle cases where an update might fail due to invalid or missing location data. Ensure that your data management processes account for these situations.

This setup should ensure that your `Customers` table maintains data integrity with a required `SearchLocation` column, supporting efficient and consistent use of location data in your application. Let me know if you have further questions or need additional assistance!

Okay, looks like this worked. Except I had to use the first SQL code you provided for the `SearchColumn` field addition. It seems that BigQuery doesn't allow you to add a required field after a table is created.



That's correct; in BigQuery, you can't add a column with a `NOT NULL` constraint to an existing table. When you add a new column, it must initially be nullable, and you can enforce the requirement at the application level or by ensuring all rows have valid data before altering the schema to enforce constraints through different methods.

Enforcing Required Fields in BigQuery

Since BigQuery doesn't support altering an existing column to be `NOT NULL`, here are a few strategies to ensure the field is always populated:

1. **Application Logic:** Ensure that your application or scripts that insert data into BigQuery always include a value for the `SearchLocation` column. This is typically managed by input validation and logic checks before data insertion.

2. Data Loading and Validation:

- When loading data into BigQuery, perform a pre-load validation step to ensure that all required fields, such as `SearchLocation`, are present and correctly formatted.
- Use scheduled queries to identify and handle any records that might be missing required data. For example, you can set up periodic checks to ensure there are no `NULL` values in the `SearchLocation` column.

3. Using Views:

- Create a view that filters out records with missing `SearchLocation` values. This view can then be used in applications that require guaranteed presence of this field.
- Example of creating a view:

sql

```
CREATE OR REPLACE VIEW `agency seo tools
318618.CustomerGbpCompetitors.ValidCustomers` AS
SELECT *
FROM `agency seo tools 318618.CustomerGbpCompetitors.Customers`
WHERE SearchLocation IS NOT NULL;
```

4. Automated Alerts:

Set up alerts or logs to track any insertions that result in a `NULL` value for required fields, so you can take corrective action promptly.

5. Handling Default Values:

If there's a common fallback location, you could set a default value during data loading if none is provided, though this requires thoughtful handling to avoid misleading data.

These strategies help maintain data integrity in BigQuery by ensuring that critical fields like `SearchLocation` are always populated, even if the database schema itself doesn't enforce non-null constraints directly. Let me know if there's anything else you'd like to explore or implement!



docs.dataforseo.com-serpmapsliveadvanced DataForSEO API v3.pdf

PDF

Great, now let's do a test integration of the DataForSEO API. We'll need to use the SERP API. More specifically, the Live Google Maps SERP API for performing a live search from a location and returning the results from the API. The API says that the endpoint we need to use is: POST https://api.dataforseo.com/v3/serp/google/maps/task_post I've attached the API documentation here. Here is sample code provided by DataForSEO: # Instead of 'login' and 'password' use your credentials from <https://app.dataforseo.com/api-dashboard> \ login="login" \ password="password" \ cred="\$(printf \${login}:\${password} | base64)" \ curl --location --request POST "https://api.dataforseo.com/v3/serp/google/maps/live/advanced" \ --header "Authorization: Basic \${cred}" \ --header "Content-Type: application/json" \ --data-raw "[{ \"language_code\": \"en\", \"location_code\": 2840, \"keyword\": \"albert einstein\" }]" Our API Login is: [REDACTED] API Password: [REDACTED] Please update the python script to perform a search for "[REDACTED]" from the search location "Toronto,Ontario,Canada". Use static values for now instead of database values.



Let's integrate the DataForSEO API to perform a search using the Live Google Maps SERP API. We'll write a Python script that sends a POST request to the API to search for "criminal lawyer toronto" from the location "Toronto,Ontario,Canada". We'll use your provided credentials for authentication.

Python Script for API Integration

Here's how you can update your Cloud Function to perform this API call:

python

```
import functions_framework
from google.cloud import bigquery
import json
import requests
from requests.auth import HTTPBasicAuth

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    # Set up your DataForSEO API credentials
    api_login = ████████████████████████████████████████████████████████████
    api_password = ████████████████████████████████████████████████████████████

    # Define the API endpoint
    api_url = "https://api.dataforseo.com/v3/serp/google/maps/task_post"

    # Set up the payload for the request
    payload = [
        {
            "language code": "en",
            "location name": "Toronto, Ontario, Canada",
            "keyword": "██████████████████████████████████████████████████████████"
        }
    ]

    # Send a POST request to the DataForSEO API
    try:
        response = requests.post(
            api_url,
            headers={"Content-Type": "application/json"},
            auth=HTTPBasicAuth(api_login, api_password),
            json=payload
        )

    # Check if the request was successful
    if response.status_code == 200:
        result = response.json()
        return json.dumps(result, indent=2), 200, {'Content-Type':
```

```
'application/json'}
    else:
        return f"Error: {response.status_code} - {response.text}",
response.status_code

except Exception as e:
    return f"An error occurred: {str(e)}", 500
```

Explanation

- **Authentication:** Uses HTTP Basic Authentication with your login and password encoded as credentials.
- **API Endpoint:** The POST request is sent to the specified DataForSEO endpoint for Google Maps SERP.
- **Payload:** Contains the search parameters, including `language_code`, `location_name`, and `keyword`.
- **Response Handling:** The response is checked for a successful status code (200). If successful, it returns the JSON result. If not, it returns an error message with the status code.

Testing the Function

1. **Deploy the Function:** Deploy this updated script to your Cloud Function environment.
2. **Invoke the Function:** Send a request to the function's HTTP URL and check the JSON output to see if the API returns the expected results.
3. **Review the Output:** The JSON response should include the search results for the specified keyword and location.

This setup should help you test the integration with DataForSEO and verify that you can successfully retrieve search results using their API. If you encounter any issues, feel free to reach out for further assistance!

Sorry, I realize I wasn't sufficiently clear about my request. I wanted to update the existing python script that connects to the BigQuery database with code that will perform the sample DataForSEO API request. However, I realize that including the code in the same file that contains the BigQuery code may not be the best way to set up the code. Should we create a new python file that is used only for performing API tasks? If so, how can we set up that file and reference it from the main python script file (main.py)?



Organizing your code into multiple files can help keep your codebase clean and maintainable. For your case, you can create a separate Python module to handle the API tasks, such as interacting with the DataForSEO API, and then reference this module in your main Cloud Function script.

Steps to Organize Your Code

1. Create a New Python Module for API Tasks

Create a new Python file (e.g., `api_tasks.py`) that will contain functions for interacting with the DataForSEO API.

2. Define API Functionality in the New Module

Implement functions in `api_tasks.py` to handle API requests and processing.

3. Import the API Module in Your Main Script

In your main Cloud Function script (`main.py`), import the `api_tasks` module and use its functions as needed.

Example Implementation

Step 1: Create `api_tasks.py`

Create a file named `api_tasks.py` in the same directory as your main script. Add the following code to handle the DataForSEO API requests:

python

```
import requests
from requests.auth import HTTPBasicAuth
import json

# Function to perform a search using the DataForSEO API
def perform_dataforseo_search(keyword, location_name):
    # Set up your DataForSEO API credentials
    api_login = [REDACTED]
    api_password = [REDACTED]

    # Define the API endpoint
    api_url = "https://api.dataforseo.com/v3/serp/google/maps/task_post"

    # Set up the payload for the request
    payload = [
        {
            "language_code": "en",
            "location_name": location_name,
            "keyword": keyword
        }
    ]

    # Send a POST request to the DataForSEO API
    response = requests.post(
        api_url,
        headers={"Content-Type": "application/json"},
        auth=HTTPBasicAuth(api_login, api_password),
        json=payload
    )

    # Check if the request was successful
    if response.status_code == 200:
        return response.json()
    else:
        raise Exception(f"Error: {response.status_code} - {response.text}")
```

Step 2: Update `main.py`

In your main Cloud Function script, import the `api_tasks` module and use the `perform_dataforseo_search` function:

python

```
import functions_framework
from google.cloud import bigquery
import json
import api_tasks # Import the api_tasks module

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    # Instantiate a BigQuery client
    client = bigquery.Client(project="agency-seo-tools-318618")

    # Define the query to join Customers and CustomerQueries
    query = """
        SELECT
            c.CustomerName,
            q.Query,
            c.SearchLocation
        FROM
            `agency-seo-tools-318618.CustomerGbpCompetitors.Customers` c
    JOIN
        `agency-seo-tools-318618.CustomerGbpCompetitors.CustomerQueries` q
    ON
        c.CustomerID = q.CustomerID
    ORDER BY
        c.CustomerName, q.Query;
    """

    try:
        # Perform the query
        query_job = client.query(query) # Make an API request
        results = query_job.result() # Waits for the job to complete

        # Create a dictionary to hold customers and their queries
        customer_queries = {}
        for row in results:
            customer_name = row.get("CustomerName")
            query_text = row.get("Query")
            search_location = row.get("SearchLocation")
```

```

        if customer_name not in customer_queries:
            customer_queries[customer_name] = []

        customer_queries[customer_name].append({
            "query": query_text,
            "location": search_location
        })

    # Perform a sample API request for "██████████"
    # This is just a static test; you can loop through customer_queries
    to perform dynamic requests.
    api_result = api_tasks.perform_dataforseo_search("██████████
██████████", "Toronto,Ontario,Canada")

    # Add the API result to the output for demonstration purposes
    output = {
        "customer_queries": customer_queries,
        "api_result": api_result
    }

    # Return the results as JSON
    return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}

except Exception as e:
    return f"An error occurred: {str(e)}", 500

```

Explanation

- **api_tasks.py:** Contains the `perform_dataforseo_search` function, which handles API requests to DataForSEO.
- **Modularization:** Separating API tasks into their own module keeps your code organized and makes it easier to manage and test.
- **Importing Modules:** You import `api_tasks` in your `main.py` script and call the `perform_dataforseo_search` function as needed.

Testing

1. **Deploy Both Files:** Ensure both `main.py` and `api_tasks.py` are deployed as part of your Cloud Function. The deployment process will package all files in the directory.
2. **Invoke the Function:** Send a request to the Cloud Function URL to see the combined results from BigQuery and DataForSEO in the JSON output.

3. **Review Logs:** Check the logs in the Cloud Console to debug any issues that may arise during execution.

This setup allows you to manage API tasks separately, facilitating maintenance and scalability of your application. If you have further questions or need additional functionality, feel free to ask!

Great. I chose a different name for the api python file. I named the file "dfseo_api.py". I changed all necessary references in the code you provided as well. When the script was tested we received an error that the request entity was too large. I believe this is because the DataForSEO API returned too much data to process or output in the test. I will provide sample output provided by the DataForSEO API documentation below. We only want to output the names of the business found in the "items" JSON array that is found in the data that the API returns. Also, we need to update the API request in the dfseo_api.py file to use the following URL: <https://api.dataforseo.com/v3/serp/google/maps/live/advanced>. We also need to specifically request the API to return the top 6 results by using the "depth" api parameter. Sample Output: { "version": "0.1.20230705", "status_code": 20000, "status_message": "Ok.", "time": "4.0977 sec.", "cost": 0.002, "tasks_count": 1, "tasks_error": 0, "tasks": [{ "id": "08021555-1535-0139-0000-462b80e927f7", "status_code": 20000, "status_message": "Ok.", "time": "4.0299 sec.", "cost": 0.002, "result_count": 1, "path": ["v3", "serp", "google", "maps", "live", "advanced"], "data": { "api": "serp", "function": "live", "se": "google", "se_type": "maps", "language_code": "en", "location_code": 2840, "keyword": "albert einstein", "depth": 3, "device": "desktop", "os": "windows" }, "result": [{ "keyword": "albert einstein", "type": "maps", "se_domain": "google.com", "location_code": 2840, "language_code": "en", "check_url": "https://google.com/maps/search/albert+einstein/@37.09024,-95.712891,4z?hl=en&gl=US&uule=w+CAIQIFISCQs2MuSEtepUEUK33kOSuTsc", "datetime": "2023-08-02 12:55:43 +00:00", "spell": null, "item_types": ["maps_search"], "se_results_count": 0, "items_count": 3, "items": [{ "type": "maps_search", "rank_group": 1, "rank_absolute": 1, "domain": null, "title": "Albert Einstein Statue", "url": null, "contact_url": null, "rating": null, "rating_distribution": null, "snippet": "6700 NW 35th St, Bethany, OK 73008", "address": "6700 NW 35th St, Bethany, OK 73008", "address_info": { "borough": "Central Oklahoma City", "address": "6700 NW 35th St", "city": "Bethany", "zip": "73008", "region": "Oklahoma", "country_code": "US" }, "place_id": "ChIJSe0feYkFsocRajueupcvpas", "phone": null, "main_image": "https://lh5.googleusercontent.com/p/AF1QipOfM59_liNdL9hmEDbU0_K1uTSv4psb

h541-k-no", "total_photos": 12, "category": "Tourist attraction",
"additional_categories": null, "category_ids": ["tourist_attraction"],
"work_hours": { "timetable": null, "current_status": "close" }, "feature_id":
"0x87b20589791fed49:0xaba52f97ba9e3b6a", "cid":
"12368344280360434538", "latitude": 35.5064351, "longitude":
-97.6317255, "is_claimed": false, "local_justifications": null,
"is_directory_item": false, "price_level": null, "hotel_rating": null }, { "type":
"maps_search", "rank_group": 2, "rank_absolute": 2, "domain":
"www.nasonline.org", "title": "Albert Einstein Memorial", "url":
"http://www.nasonline.org/about-nas/visiting-nas/nas-building/the-einstein-
memorial.html", "contact_url": null, "rating": { "rating_type": "Max5", "value":
4.7, "votes_count": 1283, "rating_max": null }, "rating_distribution": { "1": 4,
"2": 8, "3": 51, "4": 194, "5": 1026 }, "snippet": "2101 Constitution Ave. NW,
Washington, DC 20418", "address": "2101 Constitution Ave. NW,
Washington, DC 20418", "address_info": { "borough": "Northwest
Washington", "address": "2101 Constitution Ave. NW", "city":
"Washington", "zip": "20418", "region": "District of Columbia",
"country_code": "US" }, "place_id": "ChIJJZaaKqm3t4kRbBG17Vcv8Rg",
"phone": null, "main_image":
"https://lh5.googleusercontent.com/p/AF1QipPyOiGml7eIl8LgiYGtPh4wzI7g8GP3fZs
h306-k-no", "total_photos": 2501, "category": "Sculpture",
"additional_categories": ["Monument"], "category_ids": ["sculpture",
"monument"], "work_hours": { "timetable": { "sunday": [{ "open": { "hour":
0, "minute": 0 }, "close": { "hour": 0, "minute": 0 } }], "monday": [{ "open": {
"hour": 0, "minute": 0 }, "close": { "hour": 0, "minute": 0 } }], "tuesday": [{
"open": { "hour": 0, "minute": 0 }, "close": { "hour": 0, "minute": 0 } }],
"wednesday": [{ "open": { "hour": 0, "minute": 0 }, "close": { "hour": 0,
"minute": 0 } }], "thursday": [{ "open": { "hour": 0, "minute": 0 }, "close": {
"hour": 0, "minute": 0 } }], "friday": [{ "open": { "hour": 0, "minute": 0 },
"close": { "hour": 0, "minute": 0 } }], "saturday": [{ "open": { "hour": 0,
"minute": 0 }, "close": { "hour": 0, "minute": 0 } }] }, "current_status": "open"
}, "feature_id": "0x89b7b7a92a9a9625:0x18f12f57edb5116c", "cid":
"1797269780994265452", "latitude": 38.8924028, "longitude":
-77.0484139, "is_claimed": false, "local_justifications": null,
"is_directory_item": false, "price_level": null, "hotel_rating": null }, { "type":
"maps_search", "rank_group": 3, "rank_absolute": 3, "domain":
"www.alberteinsteinigroup.com", "title": "The Albert Einstein Group, Inc.",
"url": "http://www.alberteinsteinigroup.com/", "contact_url": null, "rating": {
"rating_type": "Max5", "value": 4, "votes_count": 4, "rating_max": null },
"rating_distribution": { "1": 1, "2": 0, "3": 0, "4": 0, "5": 3 }, "snippet": "8710
Converse Business Cir, Converse, TX 78109", "address": "8710 Converse
Business Cir, Converse, TX 78109", "address_info": { "borough":

```
"Northeast Side", "address": "8710 Converse Business Cir", "city":  
"Converse", "zip": "78109", "region": "Texas", "country_code": "US" },  
"place_id": "ChIJ5QKOJR6KXIYRm3-tvYC0RuE", "phone": "+1833-463-  
3325", "main_image":  
"https://lh5.googleusercontent.com/p/AF1QipMwYCFGMituQ_CohAgfNF6pQPNjt-  
b-FR96b7xc=w416-h240-k-no", "total_photos": 18, "category": "Business  
to business service", "additional_categories": null, "category_ids": [  
"business_to_business_service" ], "work_hours": { "timetable": { "sunday":  
null, "monday": [ { "open": { "hour": 8, "minute": 0 }, "close": { "hour": 17,  
"minute": 0 } } ], "tuesday": [ { "open": { "hour": 8, "minute": 0 }, "close": {  
"hour": 17, "minute": 0 } } ], "wednesday": [ { "open": { "hour": 8, "minute": 0  
}, "close": { "hour": 17, "minute": 0 } } ], "thursday": [ { "open": { "hour": 8,  
"minute": 0 }, "close": { "hour": 17, "minute": 0 } } ], "friday": [ { "open": {  
"hour": 8, "minute": 0 }, "close": { "hour": 17, "minute": 0 } } ], "saturday": [ {  
"open": { "hour": 8, "minute": 0 }, "close": { "hour": 12, "minute": 0 } } ] } },  
"current_status": "close" }, "feature_id":  
"0x865c8a1e258e02e5:0xe146b480bdad7f9b", "cid":  
"16232860371934609307", "latitude": 29.515755199999997, "longitude":  
-98.3017169, "is_claimed": true, "local_justifications": null,  
"is_directory_item": false, "price_level": null, "hotel_rating": null } ] ] ] ] ] }
```



To address the issue of large response sizes, let's refine the code to extract only the names of businesses from the `items` array returned by the DataForSEO API. We'll also update the `dfseo_api.py` module to use the correct URL and include the `depth` parameter to limit the number of results.

Updated `dfseo_api.py`

Let's modify the `dfseo_api.py` to handle the API request and response processing:

python

```
import requests
from requests.auth import HTTPBasicAuth
import json

def perform_dataforseo_search(keyword, location_name):
    # Set up your DataForSEO API credentials
    api_login = [REDACTED]
    api_password = [REDACTED]

    # Define the API endpoint for the advanced live Google Maps search
    api_url =
"https://api.dataforseo.com/v3/serp/google/maps/live/advanced"

    # Set up the payload for the request with the depth parameter
    payload = [
        {
            "language_code": "en",
            "location_name": location_name,
            "keyword": keyword,
            "depth": 6 # Limit the number of results to the top 6
        }
    ]

    # Send a POST request to the DataForSEO API
    response = requests.post(
        api_url,
        headers={"Content-Type": "application/json"},
        auth=HTTPBasicAuth(api_login, api_password),
        json=payload
    )

    # Check if the request was successful
    if response.status_code == 200:
        return response.json()
    else:
        raise Exception(f"Error: {response.status_code} - {response.text}")

def extract_business_names(api_response):
    # Extract business names from the API response
    business_names = []
    try:
        tasks = api_response.get("tasks", [])
        for task in tasks:
            result = task.get("result", [])
```

```
    for res in result:
        items = res.get("items", [])
        for item in items:
            business_names.append(item.get("title"))
except Exception as e:
    raise Exception(f"Error extracting business names: {str(e)}")

return business_names
```

Explanation

- **Depth Parameter:** The `depth` parameter is set to 6, limiting the API to return only the top 6 results, reducing response size.
- **Extract Business Names:** A new function `extract_business_names` is added to parse the response and collect the names of businesses from the `items` array.

Updated `main.py`

Let's update `main.py` to use these functions and handle the extracted business names:

python

```
import functions_framework
from google.cloud import bigquery
import json
import dfseo_api # Import the dfseo_api module

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    # Instantiate a BigQuery client
    client = bigquery.Client(project="agency-seo-tools-318618")

    # Define the query to join Customers and CustomerQueries
    query = """
        SELECT
            c.CustomerName,
            q.Query,
            c.SearchLocation
        FROM
            `agency-seo-tools-318618.CustomerGbpCompetitors.Customers` c
        JOIN
            `agency-seo-tools-318618.CustomerGbpCompetitors.CustomerQueries` q
        ON
            c.CustomerID = q.CustomerID
        ORDER BY
            c.CustomerName, q.Query;
    """

    try:
        # Perform the query
        query_job = client.query(query) # Make an API request
        results = query_job.result() # Waits for the job to complete

        # Create a dictionary to hold customers and their queries
        customer_queries = {}
        for row in results:
            customer_name = row.get("CustomerName")
            query_text = row.get("Query")
```



```

search_location = row.get("SearchLocation")

if customer_name not in customer_queries:
    customer_queries[customer_name] = []

customer_queries[customer_name].append({
    "query": query_text,
    "location": search_location
})

# Perform a sample API request for "████████████████████"
api_response = dfseo_api.perform_dataforseo_search("████████████████████
██████████", "Toronto,Ontario,Canada")
business_names = dfseo_api.extract_business_names(api_response)

# Add the business names to the output for demonstration purposes
output = {
    "customer_queries": customer_queries,
    "business_names": business_names
}

# Return the results as JSON
return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}

except Exception as e:
    return f"An error occurred: {str(e)}", 500

```

Explanation

- **API Request and Parsing:** The `perform_dataforseo_search` function sends the API request, and the `extract_business_names` function extracts the business names from the response.
- **JSON Output:** The response now includes only the business names, reducing the size of the data returned and focusing on the relevant information.

Testing

1. **Deploy Both Files:** Ensure both `main.py` and `dfseo_api.py` are deployed as part of your Cloud Function package.
2. **Invoke the Function:** Send a request to your Cloud Function URL and verify that the JSON output includes the business names extracted from the API response.
3. **Review Logs:** Check the logs in the Google Cloud Console to troubleshoot any issues and verify successful execution.

By focusing on extracting only the business names, you reduce the amount of data processed and returned, making the function more efficient and tailored to your needs. If you have further questions or require additional features, feel free to ask!

I want to rename the python file from `dfseo_api.py` to `seo_api.py` because it's possible we may use a different API in the future and I'd like to make the code naming API agnostic. Can we update the code to use API agnostic names. For example, instead of the function name "`perform_dataforseo_search`" I'd like to use "`perform_api_maps_search`".



Certainly! Renaming your file and functions to use more generic names is a good approach for future-proofing your code. Here's how you can update the code to use API-agnostic names.

Step 1: Rename `dfseo_api.py` to `seo_api.py`

Rename the file to `seo_api.py` and update the function names within it:

python

```
import requests
from requests.auth import HTTPBasicAuth
import json

def perform_api_maps_search(keyword, location_name):
    # Set up your API credentials
    api_login = ████████████████████████████████████████████████████
    api_password = ████████████████████████████████████████████████████

    # Define the API endpoint for the advanced live Google Maps search
    api_url =
"https://api.dataforseo.com/v3/serp/google/maps/live/advanced"

    # Set up the payload for the request with the depth parameter
    payload = [
        {
            "language_code": "en",
            "location_name": location_name,
            "keyword": keyword,
            "depth": 6 # Limit the number of results to the top 6
        }
    ]

    # Send a POST request to the API
    response = requests.post(
        api_url,
        headers={"Content-Type": "application/json"},
        auth=HTTPBasicAuth(api_login, api_password),
        json=payload
    )

    # Check if the request was successful
    if response.status_code == 200:
        return response.json()
    else:
        raise Exception(f"Error: {response.status_code} - {response.text}")

def extract_business_names(api_response):
    # Extract business names from the API response
    business_names = []
    try:
        tasks = api_response.get("tasks", [])
        for task in tasks:
            result = task.get("result", [])
```

```
        for res in result:
            items = res.get("items", [])
            for item in items:
                business_names.append(item.get("title"))
    except Exception as e:
        raise Exception(f"Error extracting business names: {str(e)}")

    return business_names
```

Step 2: Update `main.py` to Use the Renamed Module and Functions

Update your `main.py` script to import and use the renamed module and functions:

python

```
import functions_framework
from google.cloud import bigquery
import json
import seo_api # Import the seo_api module

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    # Instantiate a BigQuery client
    client = bigquery.Client(project="agency-seo-tools-318618")

    # Define the query to join Customers and CustomerQueries
    query = """
        SELECT
            c.CustomerName,
            q.Query,
            c.SearchLocation
        FROM
            `agency-seo-tools-318618.CustomerGbpCompetitors.Customers` c
        JOIN
            `agency-seo-tools-318618.CustomerGbpCompetitors.CustomerQueries` q
        ON
            c.CustomerID = q.CustomerID
        ORDER BY
            c.CustomerName, q.Query;
    """

    try:
        # Perform the query
        query_job = client.query(query) # Make an API request
        results = query_job.result() # Waits for the job to complete

        # Create a dictionary to hold customers and their queries
        customer_queries = {}
        for row in results:
            customer_name = row.get("CustomerName")
            query_text = row.get("Query")
```

```

        search_location = row.get("SearchLocation")

        if customer_name not in customer_queries:
            customer_queries[customer_name] = []

        customer_queries[customer_name].append({
            "query": query_text,
            "location": search_location
        })

        # Perform a sample API request for "██████████"
        api_response = seo_api.perform_api_maps_search("██████████
██████████", "Toronto,Ontario,Canada")
        business_names = seo_api.extract_business_names(api_response)

        # Add the business names to the output for demonstration purposes
        output = {
            "customer_queries": customer_queries,
            "business_names": business_names
        }

        # Return the results as JSON
        return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}

    except Exception as e:
        return f"An error occurred: {str(e)}", 500

```

Key Changes

- **Module Name:** Changed `dfseo_api.py` to `seo_api.py`.
- **Function Names:** Updated function names to be more generic (`perform_api_maps_search`).
- **Imports:** Updated the import statement in `main.py` to reflect the new module name.

Testing

1. **Rename the File:** Ensure you rename the file in your local or development environment to `seo_api.py`.
2. **Deploy:** Deploy both `main.py` and `seo_api.py` as part of your Cloud Function package.
3. **Invoke the Function:** Test the function via the HTTP URL and verify the JSON output to ensure it correctly lists the business names.

4. **Check Logs:** Use the Google Cloud Console to check logs for any issues or to confirm successful execution.

This setup makes your code more flexible and ready to accommodate changes in the API provider or further expansions in functionality. If you have any further questions or need additional support, feel free to ask!

Can we extract from the main.py file, all code that is related to BigQuery functions and move that into a separate python file called "big_query.py"? I'd like the main.py file to be limited to procedural steps/actions and place all code for performing those actions in separate files.



Certainly! Organizing your code by separating concerns into different modules is a great way to improve maintainability and clarity. Let's move all the BigQuery-related code into a new file called `big_query.py` and leave the procedural logic in `main.py`.

Step 1: Create `big_query.py`

Create a new file named `big_query.py` that will contain all the BigQuery-related functions. Here's how you can define the functions for querying customer data:

python

```
from google.cloud import bigquery

def get_customer_queries(project_id):
    """Fetches customer queries and search locations from BigQuery."""
    client = bigquery.Client(project=project_id)

    query = """
        SELECT
            c.CustomerName,
            q.Query,
            c.SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers` c
        JOIN
            `{}`.CustomerGbpCompetitors.CustomerQueries` q
        ON
            c.CustomerID = q.CustomerID
        ORDER BY
            c.CustomerName, q.Query;
    """.format(project_id, project_id)

    query_job = client.query(query) # Make an API request
    results = query_job.result() # Waits for the job to complete

    # Create a dictionary to hold customers and their queries
    customer_queries = {}
    for row in results:
        customer_name = row.get("CustomerName")
        query_text = row.get("Query")
        search_location = row.get("SearchLocation")

        if customer_name not in customer_queries:
            customer_queries[customer_name] = []

        customer_queries[customer_name].append({
            "query": query_text,
            "location": search_location
        })

    return customer_queries
```


Step 2: Update `main.py`

Update `main.py` to import and use the BigQuery functions from `big_query.py`:

python

```
import functions_framework
import json
import seo api # Import the seo api module
import big query # Import the big query module

@functions framework.http
def script init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"

        # Fetch customer queries from BigQuery
        customer_queries = big_query.get_customer_queries(project_id)

        # Perform a sample API request for "████████████████████"
        api response = seo api.perform api maps search("████████████████████", "Toronto,Ontario,Canada")
        business names = seo api.extract business names(api response)

        # Add the business names to the output for demonstration purposes
        output = {
            "customer queries": customer_queries,
            "business names": business names
        }

        # Return the results as JSON
        return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}

    except Exception as e:
        return f"An error occurred: {str(e)}", 500
```

Explanation

- **big_query.py**: Contains the `get_customer_queries` function, which handles all interactions with BigQuery to retrieve customer data.
- **main.py**: Now focuses on procedural logic, orchestrating calls to functions in `seo_api.py` and `big_query.py`.

Testing

1. **Create and Organize Files**: Ensure `big_query.py`, `seo_api.py`, and `main.py` are in the same directory and contain the correct code as shown.
2. **Deploy the Function**: Deploy all three files as part of your Cloud Function package.
3. **Invoke the Function**: Use the Cloud Function URL to test the endpoint and verify that it returns the expected JSON output.
4. **Log Monitoring**: Check logs in the Google Cloud Console for any issues or confirmations of successful execution.

This setup modularizes your code, making it easier to manage, test, and scale in the future. If you need further assistance or have additional questions, feel free to reach out!

Great, now lets update the project code files so that we do the following: - Fetch the first customer from BigQuery and their search location - Fetch all queries for the customer - Loop through all queries for the customer - Use the API to perform each query from the customer's search location - Find the top 6 competitor business names (businesses that do not match the customer's name) - Create an array of unique business names found for all queries - Output the Customer name and then the list of unique competitor names found across all queries performed.



To implement the specified functionality, we'll update the `big_query.py` file to fetch a specific customer's data and their queries, then adjust the `main.py` to loop through the queries, perform the API calls, and collect unique competitor names.

Here's how you can structure the code to achieve this:

Step 1: Update `big_query.py`

We need to update the `big_query.py` to fetch the first customer and their queries:

python

```
from google.cloud import bigquery

def get_first_customer_with_queries(project_id):
    """Fetches the first customer and their queries from BigQuery."""
    client = bigquery.Client(project=project_id)

    # Fetch the first customer with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`
        LIMIT 1;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customer = next(customer_query_job.result(), None) # Get the first
customer

    if not customer:
        raise Exception("No customers found in the database.")

    customer_id = customer.get("CustomerID")
    customer_name = customer.get("CustomerName")
    search_location = customer.get("SearchLocation")

    # Fetch all queries for the customer
    queries_query = """
        SELECT Query
        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        WHERE CustomerID = {};
    """.format(project_id, customer_id)

    queries_job = client.query(queries_query)
    queries = [row.get("Query") for row in queries_job.result()]

    return customer_name, search_location, queries
```

Step 2: Update `main.py`

Now, let's update `main.py` to loop through the queries, call the API, and collect unique competitor names:

python

```
import functions_framework
import json
import seo_api # Import the seo api module
import big_query # Import the big query module

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"

        # Fetch the first customer and their queries
        customer_name, search_location, queries =
big_query.get_first_customer_with_queries(project_id)

        # Create a set to hold unique competitor names
        competitor_names = set()

        # Loop through all queries for the customer
        for query in queries:
            # Perform an API request for each query
            api_response = seo_api.perform_api_maps_search(query,
search_location)
            business_names = seo_api.extract_business_names(api_response)

            # Find top 6 competitors (exclude customer name)
            for name in business_names[:6]:
                if name != customer_name:
                    competitor_names.add(name)

        # Prepare the output with the customer name and unique competitors
        output = {
            "customer_name": customer_name,
            "unique_competitors": list(competitor_names)
        }

        # Return the results as JSON
```

```
        return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}

    except Exception as e:
        return f"An error occurred: {str(e)}", 500
```

Explanation

- **big_query.py:**

get_first_customer_with_queries: Fetches the first customer and their search location, along with all queries associated with that customer.

- **main.py:**

- Loops through each query for the customer.
- Calls the API using `seo_api.perform_api_maps_search`.
- Extracts the top 6 competitor names, ensuring they don't match the customer's name.
- Collects unique competitor names across all queries in a set.

Testing

1. **Deploy Both Files:** Ensure all three files (`main.py`, `big_query.py`, `seo_api.py`) are deployed together as part of your Cloud Function package.
2. **Invoke the Function:** Test the function by sending a request to the Cloud Function URL and verify that it returns the expected JSON output.
3. **Review Output:** Ensure that the output includes the customer name and a list of unique competitor names.
4. **Monitor Logs:** Use the Google Cloud Console to review logs and troubleshoot any issues that arise during execution.

This implementation efficiently loops through queries, performs API calls, and collects the desired information, keeping the code organized and maintainable. If you need further assistance or adjustments, feel free to ask!

I think this worked. However, there is a problem with the name for [REDACTED] the name should be "[REDACTED]" can you write the SQL to update this Customer table entry?



Certainly! To update the customer name for [REDACTED] in the `Customers` table, you can use the following SQL `UPDATE` statement:

sql

```
UPDATE `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`  
SET CustomerName '██████████████████████████████████████████'  
WHERE CustomerName '████████████████████';
```

Explanation

- **UPDATE Statement:** Modifies the `CustomerName` field for records where the current `CustomerName` is `'████████████████████'`.
- **SET Clause:** Assigns the new name `'██'` to the matching record.

Executing the SQL

1. BigQuery Console:

- Open the BigQuery Console.
- In the query editor, paste the SQL statement above.
- Click "Run" to execute the query and update the record.

2. Python Script (Optional):

If you prefer, you can also execute this SQL query programmatically using a Python script with the BigQuery client library.

Here's a sample Python script to execute the update query programmatically:

python

```
from google.cloud import bigquery

def update_customer_name(project_id):
    """Update the customer name for ██████████ in BigQuery."""
    client = bigquery.Client(project=project_id)

    # Define the update query
    update_query = """
        UPDATE `{}.CustomerGbpCompetitors.Customers`
        SET CustomerName = '████████████████████████████████████████'
        WHERE CustomerName = '████████████████████';
    """.format(project_id)

    # Execute the update query
    query_job = client.query(update_query) # Make an API request
    query_job.result() # Wait for the query to complete

    print("Customer name updated successfully.")

# Example usage
update_customer_name("agency-seo-tools-318618")
```

Testing

- After executing the update, you can verify the change by running a **SELECT** query to fetch the customer names and ensure the update was successful.
- Make sure there are no typos or inconsistencies in the customer name when performing the update.

This approach will ensure that the customer name is updated correctly in your BigQuery table. Let me know if you have any further questions or need additional assistance!

Great, now let's expand on the project code. We'll need a function in the `big_query.py` file that adds a competitor to the database, if the competitor doesn't already exist. The `CompetitorID` should use the `'place_id'` value that is returned from the API for that business. When determining if a

competitor exists already in the database, we should use this `place_id` value instead of the business name, as the business name could change from time to time.



To implement this functionality, we'll create a new function in the `big_query.py` file that checks if a competitor already exists in the database using the `place_id`. If the competitor does not exist, it will insert the competitor into the `Competitors` table using the `place_id` as the `CompetitorID`.

Here's how you can implement the function:

Step 1: Update `big_query.py`

Add the following function to `big_query.py` to handle checking for existing competitors and adding new ones:

python

```
from google.cloud import bigquery

def get_bigquery_client(project_id):
    """Returns a BigQuery client for the specified project."""
    return bigquery.Client(project=project_id)

def get_first_customer_with_queries(project_id):
    """Fetches the first customer and their queries from BigQuery."""
    client = get_bigquery_client(project_id)

    # Fetch the first customer with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`
        LIMIT 1;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customer = next(customer_query_job.result(), None) # Get the first
customer

    if not customer:
        raise Exception("No customers found in the database.")

    customer_id = customer.get("CustomerID")
    customer_name = customer.get("CustomerName")
    search_location = customer.get("SearchLocation")

    # Fetch all queries for the customer
    queries_query = """
        SELECT Query
        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        WHERE CustomerID = {};
    """.format(project_id, customer_id)

    queries_job = client.query(queries_query)
    queries = [row.get("Query") for row in queries_job.result()]

    return customer_name, search_location, queries
```

```

def add_competitor_if_not_exists(project_id, competitor_name, place_id,
review_count=None, average_rating=None):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client(project_id)

    # Check if the competitor already exists using place_id
    check_query = """
        SELECT CompetitorID
        FROM `{}`.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = '{}';
    """.format(project_id, place_id)

    check_job = client.query(check_query)
    competitor = next(check_job.result(), None)

    if competitor:
        print(f"Competitor '{competitor_name}' already exists with
CompetitorID '{place_id}'.")
        return

    # Insert the new competitor into the database
    insert_query = """
        INSERT INTO `{}`.CustomerGbpCompetitors.Competitors` (CompetitorID,
CompetitorName, ReviewCount, AverageRating)
        VALUES ('{}', '{}', {}, {});
    """.format(
        project_id,
        place_id,
        competitor_name.replace("'", "'"), # Escape single quotes in
names
        review_count if review_count is not None else 'NULL',
        average_rating if average_rating is not None else 'NULL'
    )

    insert_job = client.query(insert_query)
    insert_job.result() # Wait for the query to complete

    print(f"Competitor '{competitor_name}' added with CompetitorID
'{}'.".format(place_id))

```

Explanation

- **get_bigquery_client:** A helper function to create a BigQuery client, making it easier to use in other functions.

- **add_competitor_if_not_exists:**

- **Check Competitor:** Uses a `SELECT` query to check if a competitor with the given `place id` already exists in the `Competitors` table.
- **Insert Competitor:** If the competitor does not exist, it inserts a new entry into the `Competitors` table with the provided details. The `CompetitorID` is set to the `place id`, which uniquely identifies the business.
- **SQL Escape:** Ensures that competitor names with single quotes are safely handled by escaping them.

Step 2: Update `main.py`

Update `main.py` to call the `add_competitor_if_not_exists` function during the process:

python

```
import functions_framework
import json
import seo api # Import the seo api module
import big query # Import the big query module

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"

        # Fetch the first customer and their queries
        customer_name, search_location, queries =
big_query.get_first_customer_with_queries(project_id)

        # Create a set to hold unique competitor names
        competitor_names = set()

        # Loop through all queries for the customer
        for query in queries:
            # Perform an API request for each query
            api_response = seo_api.perform_api_maps_search(query,
search_location)
            items = api_response.get("tasks", [])[0].get("result", [])[
0].get("items", [])

            # Find top 6 competitors (exclude customer name)
            for item in items[:6]:
                business_name = item.get("title")
                place_id = item.get("place_id")
                review_count = item.get("rating", {}).get("votes_count")
                average_rating = item.get("rating", {}).get("value")

                if business_name != customer_name:
                    competitor_names.add(business_name)
                    # Add competitor to database if it doesn't already
exist
```

```

        big_query.add_competitor_if_not_exists(
            project_id,
            business_name,
            place_id,
            review_count,
            average_rating
        )

    # Prepare the output with the customer name and unique competitors
    output = {
        "customer name": customer_name,
        "unique competitors": list(competitor_names)
    }

    # Return the results as JSON
    return json.dumps(output, indent=2), 200, {'Content Type':
'application/json'}

except Exception as e:
    return f"An error occurred: {str(e)}", 500

```

Explanation

- **API Response Handling:** Extracts `place_id`, `review_count`, and `average_rating` from each business in the top 6 results.
- **Add Competitor:** Uses `big_query.add_competitor_if_not_exists` to ensure each competitor is added to the database if not already present.

Testing

1. **Deploy Both Files:** Ensure all files (`main.py`, `big_query.py`, `seo_api.py`) are deployed as part of your Cloud Function package.
2. **Invoke the Function:** Test the function by sending a request to the Cloud Function URL and verify that it returns the expected JSON output and logs.
3. **Check the Database:** Use the BigQuery Console to confirm that new competitors are correctly added to the `Competitors` table.

This setup will efficiently manage the competitor data in your BigQuery database, ensuring unique entries based on the `place_id`. If you need further adjustments or have additional questions, feel free to ask!

This code didn't execute successfully, I think it may have to do with the way in which competitors are being added to the database. Let me refresh your memory of the database tables pertaining to Competitors and BusinessData. Competitors: - CompetitorID (Integer, Required) -

CompetitorName (String, 255 max length, Required) BusinessData: - RecordID (Integer, Required) - RecordDate (Date, Required) - BusinessID (Integer, Required) - BusinessType (String, 20 max length, Required) - BusinessName (String, 255 max length, Required) - ReviewCount (Integer, Nullable) - AverageRating (Float, Nullable) Please note that at the moment, we are not interested in populating any data in the BusinessData table. We are only interested in adding competitors to the Competitors table. Can you update the project code files accordingly?



Certainly! Let's update the code to focus solely on adding competitors to the `Competitors` table using the structure you've outlined.

Updated `big_query.py`

We'll update the `add competitor if not exists` function to handle the `Competitors` table with `CompetitorID` as an integer and adjust the SQL accordingly.

python

```
from google.cloud import bigquery

def get_bigquery_client(project_id):
    """Returns a BigQuery client for the specified project."""
    return bigquery.Client(project=project_id)

def get_first_customer_with_queries(project_id):
    """Fetches the first customer and their queries from BigQuery."""
    client = get_bigquery_client(project_id)

    # Fetch the first customer with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`
        LIMIT 1;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customer = next(customer_query_job.result(), None) # Get the first
customer

    if not customer:
        raise Exception("No customers found in the database.")

    customer_id = customer.get("CustomerID")
    customer_name = customer.get("CustomerName")
    search_location = customer.get("SearchLocation")

    # Fetch all queries for the customer
    queries_query = """
        SELECT Query
        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        WHERE CustomerID = {};
    """.format(project_id, customer_id)

    queries_job = client.query(queries_query)
    queries = [row.get("Query") for row in queries_job.result()]

    return customer_name, search_location, queries
```

```

def add_competitor_if_not_exists(project_id, competitor_name, place_id):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client(project_id)

    # Check if the competitor already exists using place_id
    check_query = """
        SELECT CompetitorID
        FROM `{}`.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID  {};
    """.format(project_id, place_id)

    check_job = client.query(check_query)
    competitor = next(check_job.result(), None)

    if competitor:
        print(f"Competitor '{competitor_name}' already exists with
CompetitorID '{place_id}'.")
        return

    # Insert the new competitor into the database
    insert_query = """
        INSERT INTO `{}`.CustomerGbpCompetitors.Competitors` (CompetitorID,
CompetitorName)
        VALUES ( {}, '{}');
    """.format(
        project_id,
        place_id,
        competitor_name.replace("'", "'") # Escape single quotes in names
    )

    insert_job = client.query(insert_query)
    insert_job.result() # Wait for the query to complete

    print(f"Competitor '{competitor_name}' added with CompetitorID
'{place_id}'.")

```

Updated `main.py`

Ensure `main.py` is updated to handle the logic for fetching and processing competitors:

python

```
import functions_framework
import json
import seo_api # Import the seo api module
import big_query # Import the big query module

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"

        # Fetch the first customer and their queries
        customer_name, search_location, queries =
big_query.get_first_customer_with_queries(project_id)

        # Create a set to hold unique competitor names
        competitor_names = set()

        # Loop through all queries for the customer
        for query in queries:
            # Perform an API request for each query
            api_response = seo_api.perform_api_maps_search(query,
search_location)
            items = api_response.get("tasks", [])[0].get("result", [])[0].get("items", [])

            # Find top 6 competitors (exclude customer name)
            for item in items[:6]:
                business_name = item.get("title")
                place_id = item.get("place_id")

                if business_name != customer_name:
                    competitor_names.add(business_name)
                    # Add competitor to database if it doesn't already
exist

                    big_query.add_competitor_if_not_exists(
                        project_id,
```

```

        business_name,
        int(place_id)
    )

    # Prepare the output with the customer name and unique competitors
    output = {
        "customer name": customer_name,
        "unique competitors": list(competitor_names)
    }

    # Return the results as JSON
    return json.dumps(output, indent=2), 200, {'Content Type':
'application/json'}

except Exception as e:
    return f"An error occurred: {str(e)}", 500

```

Explanation

- **CompetitorID as Integer:** The `place_id` is converted to an integer, assuming `place_id` can be an integer or you have a way to convert it appropriately. Ensure `place_id` is indeed a number or can be mapped to one.
- **No BusinessData Update:** The code does not interact with the `BusinessData` table, focusing only on adding competitors to the `Competitors` table.

Testing

1. **Deploy Both Files:** Deploy `main.py` and `big_query.py` as part of your Cloud Function package.
2. **Invoke the Function:** Send a request to your Cloud Function URL and verify the JSON output.
3. **Check BigQuery:** Use the BigQuery Console to confirm that competitors are correctly added to the `Competitors` table.
4. **Logging:** Review logs in the Google Cloud Console to ensure the code executes without errors and competitors are being added.

This updated implementation should address the issues and correctly manage the addition of competitors to your BigQuery database. If you have any more questions or need further adjustments, feel free to ask!

After executing the code, we got the following error stack: [6:41:12 PM] -
 Traceback (most recent call last): File
 "/layers/google.python.pip/pip/bin/functions-framework", line 8, in
 <module> sys.exit(_cli()) File

```
"/layers/google.python.pip/pip/lib/python3.10/site-packages/click/core.py",  
line 1157, in __call__ [6:41:12 PM] - return self.main(*args, **kwargs) File  
"/layers/google.python.pip/pip/lib/python3.10/site-packages/click/core.py",  
line 1078, in main rv = self.invoke(ctx) File  
"/layers/google.python.pip/pip/lib/python3.10/site-packages/click/core.py",  
line 1434, in invoke return ctx.invoke(self.callback, **ctx.params) File  
"/layers/google.python.pip/pip/lib/python3.10/site-packages/click/core.py",  
line 783, in invoke [6:41:12 PM] - return __callback(*args, **kwargs) File  
"/layers/google.python.pip/pip/lib/python3.10/site-  
packages/functions_framework/_cli.py", line 36, in _cli app =  
create_app(target, source, signature_type) File  
"/layers/google.python.pip/pip/lib/python3.10/site-  
packages/functions_framework/__init__.py", line 395, in create_app raise  
e from None File "/layers/google.python.pip/pip/lib/python3.10/site-  
packages/functions_framework/__init__.py", line 376, in create_app  
spec.loader.exec_module(source_module) File "<frozen  
importlib._bootstrap_external>", line 883, in exec_module File "<frozen  
importlib._bootstrap>", line 241, in _call_with_frames_removed File  
"/workspace/main.py", line 4, in <module> [6:41:12 PM] - import big_query  
# Import the big_query module File "/workspace/big_query.py", line 49_  
check_query = "" ^ SyntaxError: unterminated triple-quoted string literal  
(detected at line 51)
```



The error message indicates that there is a syntax error in your `big_query.py` file due to an unterminated triple-quoted string literal. This typically occurs when a multi-line string is not properly closed with triple quotes (`"""`). Let's correct this error.

Corrected `big_query.py`

Here is the corrected version of the `big_query.py` file:

python

```
from google.cloud import bigquery

def get_bigquery_client(project_id):
    """Returns a BigQuery client for the specified project."""
    return bigquery.Client(project=project_id)

def get_first_customer_with_queries(project_id):
    """Fetches the first customer and their queries from BigQuery."""
    client = get_bigquery_client(project_id)

    # Fetch the first customer with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`
        LIMIT 1;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customer = next(customer_query_job.result(), None) # Get the first
customer

    if not customer:
        raise Exception("No customers found in the database.")

    customer_id = customer.get("CustomerID")
    customer_name = customer.get("CustomerName")
    search_location = customer.get("SearchLocation")

    # Fetch all queries for the customer
    queries_query = """
        SELECT Query
        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        WHERE CustomerID = {};
    """.format(project_id, customer_id)

    queries_job = client.query(queries_query)
    queries = [row.get("Query") for row in queries_job.result()]

    return customer_name, search_location, queries
```

```

def add_competitor_if_not_exists(project_id, competitor_name, place_id):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client(project_id)

    # Check if the competitor already exists using place_id
    check_query = """
        SELECT CompetitorID
        FROM `{}`.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = {};
    """.format(project_id, place_id)

    check_job = client.query(check_query)
    competitor = next(check_job.result(), None)

    if competitor:
        print(f"Competitor '{competitor_name}' already exists with
CompetitorID '{place_id}'.")
        return

    # Insert the new competitor into the database
    insert_query = """
        INSERT INTO `{}`.CustomerGbpCompetitors.Competitors` (CompetitorID,
CompetitorName)
        VALUES ({}, '{}');
    """.format(
        project_id,
        place_id,
        competitor_name.replace("'", "'") # Escape single quotes in names
    )

    insert_job = client.query(insert_query)
    insert_job.result() # Wait for the query to complete

    print(f"Competitor '{competitor_name}' added with CompetitorID
'{place_id}'.")

```

Explanation

- **Triple-Quoted Strings:** Ensure all SQL queries enclosed in triple quotes (""") are properly closed.
- **CompetitorID Handling:** Double-check that `place_id` is appropriately converted to an integer where necessary and matches your data schema requirements.

Testing

1. **Deploy and Test:** Re-deploy the `main.py` and `big_query.py` files as part of your Cloud Function package.

2. **Invoke the Function:** Test the function by sending a request to the Cloud Function URL and verify the output.
3. **Check BigQuery:** Ensure competitors are correctly added to the `Competitors` table in BigQuery.
4. **Log Review:** Monitor logs in the Google Cloud Console for any further issues or confirmations of successful execution.

This correction should resolve the syntax error and enable the function to execute as expected. If there are further issues, please let me know!

This fixed the error, but when running a test, the Cloud Shell terminal shows no output from the code execution even though I think we have lines of code to output what is happening throughout the code. Can you review the code we have now and identify if there are any execution conditions that would result in no output in the terminal? If so, can you code to the files so that there is more verbose output about what the code is doing?



Let's enhance the code to include more detailed logging throughout the process to help us understand the execution flow. We'll add logging statements to track the progress and any potential issues at each step.

Updated `main.py`

Here's how you can modify `main.py` to include more verbose logging:

python

```
import functions_framework
import json
import seo_api # Import the seo api module
import big_query # Import the big query module
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"
        logging.info(f"Starting process for project ID: {project_id}")

        # Fetch the first customer and their queries
        customer name, search location, queries
        big_query.get first customer with queries(project id)
        logging.info(f"Processing customer: {customer name} with search
        location: {search location}")
        logging.info(f"Queries to process: {queries}")

        # Create a set to hold unique competitor names
        competitor names set()

        # Loop through all queries for the customer
        for query in queries:
            logging.info(f"Performing API search for query: {query}")

            # Perform an API request for each query
            api_response = seo_api.perform_api_maps_search(query,
            search_location)
            items = api_response.get("tasks", [])[0].get("result", [])[
            0].get("items", [])

            # Find top 6 competitors (exclude customer name)
```

```

        for item in items[:6]:
            business_name = item.get("title")
            place_id = item.get("place_id")

            if business_name != customer_name:
                logging.info(f"Found competitor: {business_name} with
place id: {place_id}")
                competitor_names.add(business_name)

                # Add competitor to database if it doesn't already
exist

                big_query.add_competitor_if_not_exists(
                    project_id,
                    business_name,
                    int(place_id)
                )

            # Prepare the output with the customer name and unique competitors
output = {
    "customer_name": customer_name,
    "unique_competitors": list(competitor_names)
}

            logging.info(f"Process completed for customer: {customer_name}")
            logging.info(f"Unique competitors found:
{output['unique_competitors']}")

            # Return the results as JSON
            return json.dumps(output, indent=2), 200, {'Content Type':
'application/json'}

        except Exception as e:
            logging.error(f"An error occurred: {str(e)}")
            return f"An error occurred: {str(e)}", 500

```

Updated `big_query.py`

We'll also add logging to the `big_query.py` to track database interactions:

python

```
from google.cloud import bigquery
import logging

def get_bigquery_client(project_id):
    """Returns a BigQuery client for the specified project."""
    return bigquery.Client(project=project_id)

def get_first_customer_with_queries(project_id):
    """Fetches the first customer and their queries from BigQuery."""
    client = get_bigquery_client(project_id)
    logging.info("Fetching first customer and queries from BigQuery")

    # Fetch the first customer with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`
        LIMIT 1;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customer = next(customer_query_job.result(), None) # Get the first
customer

    if not customer:
        raise Exception("No customers found in the database.")

    customer_id = customer.get("CustomerID")
    customer_name = customer.get("CustomerName")
    search_location = customer.get("SearchLocation")

    logging.info(f"Found customer: {customer_name} with ID: {customer_id}")

    # Fetch all queries for the customer
    queries_query = """
        SELECT Query
        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        WHERE CustomerID = {};
    """.format(project_id, customer_id)

    queries_job = client.query(queries_query)
```

```

queries = [row.get("Query") for row in queries_job.result()]

logging.info(f"Queries found for customer {customer_name}: {queries}")

return customer_name, search_location, queries

def add_competitor_if_not_exists(project_id, competitor_name, place_id):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client(project_id)

    # Check if the competitor already exists using place_id
    check_query = """
        SELECT CompetitorID
        FROM `{}.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = {};
    """.format(project_id, place_id)

    check_job = client.query(check_query)
    competitor = next(check_job.result(), None)

    if competitor:
        logging.info(f"Competitor '{competitor_name}' already exists with
        CompetitorID '{place_id}'.")
        return

    # Insert the new competitor into the database
    insert_query = """
        INSERT INTO `{}.CustomerGbpCompetitors.Competitors` (CompetitorID,
        CompetitorName)
        VALUES ({}, '{}');
    """.format(
        project_id,
        place_id,
        competitor_name.replace("'", "'") # Escape single quotes in names
    )

    insert_job = client.query(insert_query)
    insert_job.result() # Wait for the query to complete

    logging.info(f"Competitor '{competitor_name}' added with CompetitorID
    '{place_id}'.")

```

Key Additions

- **Logging Setup:** Configured logging at the INFO level to capture detailed execution information.
- **Informational Logs:** Added logging statements at various points in the code to trace the flow of execution, data processing, and error handling.

- **Error Logs:** Used logging to capture and report any errors encountered during execution.

Testing

1. **Deploy Both Files:** Ensure `main.py` and `big_query.py` are deployed as part of your Cloud Function package.
2. **Invoke the Function:** Test the function by sending a request to the Cloud Function URL and observe the logging output.
3. **Monitor Logs:** Use the Google Cloud Console's logging interface to monitor the detailed logs generated during execution.

These changes should help provide a clearer picture of the execution process and any potential issues. If you encounter any further problems or have specific aspects you want to enhance, let me know!

Okay, we found the problem. The database table for Competitors was set up to use an Integer value for the CompetitorID and the value being inserted to the database is the place_id which is a string.



To resolve the issue with the `CompetitorID` being set as an integer while the `place_id` is a string, you'll need to update the database schema to use a string type for the `CompetitorID` field. Here's how you can adjust the setup to accommodate this change:

Step 1: Update the Database Schema

You'll need to change the data type of the `CompetitorID` field in the `Competitors` table to a string. You can do this in the BigQuery console or using a SQL query.

SQL to Alter Table

To change the data type of the `CompetitorID` field, you can use the following SQL query:

sql

```
ALTER TABLE `agency-seo-tools-318618.CustomerGbpCompetitors.Competitors`  
ALTER COLUMN CompetitorID SET DATA TYPE STRING;
```

This command updates the `CompetitorID` to be of type `STRING`, which will allow you to store the `place_id` directly.

Step 2: Update the Code

Now that the database schema allows for `CompetitorID` as a string, ensure your code handles this properly.

Updated `big query.py`

Ensure that the `place id` is treated as a string:

python

```
from google.cloud import bigquery
import logging

def get_bigquery_client(project_id):
    """Returns a BigQuery client for the specified project."""
    return bigquery.Client(project=project_id)

def get_first_customer_with_queries(project_id):
    """Fetches the first customer and their queries from BigQuery."""
    client = get_bigquery_client(project_id)
    logging.info("Fetching first customer and queries from BigQuery")

    # Fetch the first customer with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`
        LIMIT 1;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customer = next(customer_query_job.result(), None) # Get the first
customer

    if not customer:
        raise Exception("No customers found in the database.")

    customer_id = customer.get("CustomerID")
    customer_name = customer.get("CustomerName")
    search_location = customer.get("SearchLocation")

    logging.info(f"Found customer: {customer_name} with ID: {customer_id}")

    # Fetch all queries for the customer
    queries_query = """
        SELECT Query
        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        WHERE CustomerID = {};
    """.format(project_id, customer_id)

    queries_job = client.query(queries_query)
```

```

queries = [row.get("Query") for row in queries_job.result()]

logging.info(f"Queries found for customer {customer_name}: {queries}")

return customer_name, search_location, queries

def add_competitor_if_not_exists(project_id, competitor_name, place_id):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client(project_id)

    # Check if the competitor already exists using place id
    check_query = """
        SELECT CompetitorID
        FROM `{}`.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = '{}';
    """.format(project_id, place_id)

    check_job = client.query(check_query)
    competitor = next(check_job.result(), None)

    if competitor:
        logging.info(f"Competitor '{competitor_name}' already exists with
        CompetitorID '{place_id}'.")
        return

    # Insert the new competitor into the database
    insert_query = """
        INSERT INTO `{}`.CustomerGbpCompetitors.Competitors` (CompetitorID,
        CompetitorName)
        VALUES ('{}', '{}');
    """.format(
        project_id,
        place_id,
        competitor_name.replace("'", "'") # Escape single quotes in names
    )

    insert_job = client.query(insert_query)
    insert_job.result() # Wait for the query to complete

    logging.info(f"Competitor '{competitor_name}' added with CompetitorID
    '{place_id}'.")

```

Step 3: Re-test the Function

1. **Deploy and Test:** Re-deploy your Cloud Function with the updated files.
2. **Invoke the Function:** Test by sending a request to the Cloud Function URL.
3. **Check BigQuery:** Verify that competitors are correctly added with the `place_id` as `CompetitorID`.

4. **Monitor Logs:** Review logs to ensure there are no errors and that the process is functioning as expected.

These changes should resolve the issue by allowing the `place_id` to be stored directly as a `STRING` in the database. If there are any further issues, feel free to ask!

The result of the SQL query to update the table, failed with the following error: ALTER TABLE ALTER COLUMN SET DATA TYPE requires that the existing column type (INT64) is assignable to the new type (STRING) at [2:1]



The error indicates that you cannot directly change the data type of the `CompetitorID` column from `INT64` to `STRING` because BigQuery requires the existing data to be compatible with the new type. Since `STRING` is not directly assignable from `INT64`, you'll need to take a different approach.

Alternative Approach

We'll create a new column with the desired data type, migrate the data, and then remove the old column. Here's how you can achieve this in steps:

Step 1: Add a New Column

Add a new column `CompetitorIDStr` to the table with the `STRING` data type.

```
sql
```

```
ALTER TABLE `agency-seo-tools-318618.CustomerGbpCompetitors.Competitors`  
ADD COLUMN CompetitorIDStr STRING;
```

Step 2: Migrate Data to the New Column

Copy the data from the `CompetitorID` column to the new `CompetitorIDStr` column by converting it to a string.

sql

```
UPDATE `agency-seo-tools-318618.CustomerGbpCompetitors.Competitors`  
SET CompetitorIDStr = CAST(CompetitorID AS STRING);
```

Step 3: Verify Data

Verify that the data has been correctly copied to the new column.

sql

```
SELECT CompetitorID, CompetitorIDStr  
FROM `agency-seo-tools-318618.CustomerGbpCompetitors.Competitors`  
LIMIT 10;
```

Step 4: Remove the Old Column

Once you've verified that the data has been copied correctly, you can drop the old column.

sql

```
ALTER TABLE `agency seo tools 318618.CustomerGbpCompetitors.Competitors`  
DROP COLUMN CompetitorID;
```

Step 5: Rename the New Column

Rename the new column to `CompetitorID`.

sql

```
ALTER TABLE `agency seo tools 318618.CustomerGbpCompetitors.Competitors`  
RENAME COLUMN CompetitorIDStr TO CompetitorID;
```

Update the Code

Now, you can use the updated schema with `CompetitorID` as a `STRING`.

`big_query.py`

Ensure that the `CompetitorID` handling in the code matches the updated data type:

python

```
from google.cloud import bigquery
import logging

def get_bigquery_client(project_id):
    """Returns a BigQuery client for the specified project."""
    return bigquery.Client(project=project_id)

def get_first_customer_with_queries(project_id):
    """Fetches the first customer and their queries from BigQuery."""
    client = get_bigquery_client(project_id)
    logging.info("Fetching first customer and queries from BigQuery")

    # Fetch the first customer with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`
        LIMIT 1;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customer = next(customer_query_job.result(), None) # Get the first
customer

    if not customer:
        raise Exception("No customers found in the database.")

    customer_id = customer.get("CustomerID")
    customer_name = customer.get("CustomerName")
    search_location = customer.get("SearchLocation")

    logging.info(f"Found customer: {customer_name} with ID: {customer_id}")

    # Fetch all queries for the customer
    queries_query = """
        SELECT Query
        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        WHERE CustomerID = {};
    """.format(project_id, customer_id)

    queries_job = client.query(queries_query)
```

```

queries = [row.get("Query") for row in queries_job.result()]

logging.info(f"Queries found for customer {customer_name}: {queries}")

return customer_name, search_location, queries

def add_competitor_if_not_exists(project_id, competitor_name, place_id):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client(project_id)

    # Check if the competitor already exists using place_id
    check_query = """
        SELECT CompetitorID
        FROM `{}.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = '{}';
    """.format(project_id, place_id)

    check_job = client.query(check_query)
    competitor = next(check_job.result(), None)

    if competitor:
        logging.info(f"Competitor '{competitor_name}' already exists with
        CompetitorID '{place_id}'.")
        return

    # Insert the new competitor into the database
    insert_query = """
        INSERT INTO `{}.CustomerGbpCompetitors.Competitors` (CompetitorID,
        CompetitorName)
        VALUES ('{}', '{}');
    """.format(
        project_id,
        place_id,
        competitor_name.replace("'", "'") # Escape single quotes in names
    )

    insert_job = client.query(insert_query)
    insert_job.result() # Wait for the query to complete

    logging.info(f"Competitor '{competitor_name}' added with CompetitorID
    '{place_id}'.")

```

Re-Test the Function

1. **Deploy Updated Code:** Deploy the updated code to your Cloud Function environment.
2. **Test the Function:** Send a request to the Cloud Function URL and verify the output.

3. **Check BigQuery:** Confirm that competitors are correctly added with the `place_id` as `CompetitorID`.
4. **Monitor Logs:** Review logs to ensure the function is operating correctly.

This process should resolve the schema compatibility issue and allow your code to work with `CompetitorID` as a string. Let me know if you encounter any further issues!

Okay, since we don't have any data in this table, I'll just re-create this table. I think it would be best if we also re-create the table for Customers so that the CustomerID uses the `place_id` for the business as well. I'll make these changes. Can you write the SQL to repopulate the Customers table data as well as the CustomerQueries table data? Here are the customer names again along with their `place_id` values: KCS Kitchener Cleaning Services: ChIJHeqhRy1x6lIRyGmjJ4p8PGQ [REDACTED]



Certainly! Since you are re-creating the tables, let's write the SQL queries to repopulate the `Customers` and `CustomerQueries` tables using the `place_id` as `CustomerID`.

SQL to Repopulate the `Customers` Table

Here's the SQL to insert the data into the `Customers` table:

sql

```
INSERT INTO `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`  
(CustomerID, CustomerName, SearchLocation)  
VALUES  
  ('ChIJHeqhRy1x6lIRyGmjJ4p8PGQ', 'KCS Kitchener Cleaning Services',  
  'Kitchener,Ontario,Canada'),  
  ('[REDACTED]', 'P [REDACTED]',  
  '[REDACTED]', 'Toronto,Ontario,Canada'),  
  ('[REDACTED]', '[REDACTED]', '[REDACTED]',  
  '[REDACTED]', 'Winnipeg,Manitoba,Canada'),  
  ('[REDACTED]', '[REDACTED]',  
  '[REDACTED]', 'Toronto,Ontario,Canada');
```

SQL to Repopulate the **CustomerQueries** Table

Here's the SQL to insert the query data into the **CustomerQueries** table:

- **Testing:** After populating the tables, you can test your Cloud Function to verify that it handles the new schema correctly and performs the expected operations.

This setup will ensure that your customer and query data is correctly linked to their unique identifiers in the Google Places API. Let me know if you need further assistance or additional queries!

```
Okay great. The database tables are updated to now use a String for CustomerID and CompetitorID in Customers and Competitors tables respectively. And the BusinessData table is now using a String for BusinessID. After the table updates, I ran the project code and the code generated the following error: [7:02:34 PM] - INFO:root:Starting process for project ID: agency-seo-tools-318618 [7:02:34 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running gcloud config set project or setting the GOOGLE_CLOUD_PROJECT environment variable [7:02:34 PM] - INFO:root:Fetching first customer and queries from BigQuery [7:02:35 PM] - INFO:root:Found customer: [REDACTED] ID: [REDACTED] [7:02:36 PM] - ERROR:root:An error occurred: 400 Unrecognized name: ChIjCZVw at [4:28] Location: US Job ID: be041096-3a8e-4de1-8b8b-999244362b4b
```



The error message indicates a problem with the SQL query execution related to how `CustomerID` is being referenced in the SQL statements. The issue is likely due to the way string identifiers are being concatenated into the query, which can lead to SQL injection or malformed queries if not handled properly. Let's address this issue by using parameterized queries.

Updating `big_query.py`

We'll use parameterized queries to safely insert values into the SQL statements. This approach helps avoid SQL injection issues and ensures that identifiers are correctly interpreted as strings.

Updated Code

Here's how you can modify the relevant functions in `big_query.py`:

python

```
from google.cloud import bigquery
import logging

def get_bigquery_client(project_id):
    """Returns a BigQuery client for the specified project."""
    return bigquery.Client(project=project_id)

def get_first_customer_with_queries(project_id):
    """Fetches the first customer and their queries from BigQuery."""
    client = get_bigquery_client(project_id)
    logging.info("Fetching first customer and queries from BigQuery")

    # Fetch the first customer with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`
        LIMIT 1;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customer = next(customer_query_job.result(), None) # Get the first
customer

    if not customer:
        raise Exception("No customers found in the database.")

    customer_id = customer.get("CustomerID")
    customer_name = customer.get("CustomerName")
    search_location = customer.get("SearchLocation")

    logging.info(f"Found customer: {customer_name} with ID: {customer_id}")

    # Fetch all queries for the customer using parameterized query
    queries_query = """
        SELECT Query
        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        WHERE CustomerID = @customer_id;
    """.format(project_id)

    job_config = bigquery.QueryJobConfig(
```

```

        query_parameters=[
            bigquery.ScalarQueryParameter("customer_id", "STRING",
customer_id),
        ]
    )

    queries job    client.query(queries query, job config job config)
    queries    [row.get("Query") for row in queries job.result()]

    logging.info(f"Queries found for customer {customer name}: {queries}")

    return customer name, search location, queries

def add competitor if not exists(project id, competitor name, place id):
    """Adds a competitor to the database if it does not already exist."""
    client    get bigquery client(project id)

    # Check if the competitor already exists using place_id with
parameterized query
    check_query = """
        SELECT CompetitorID
        FROM `{}`.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = @place_id;
    """.format(project_id)

    check_job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("place id", "STRING", place id),
        ]
    )

    check job    client.query(check query, job config check job config)
    competitor    next(check job.result(), None)

    if competitor:
        logging.info(f"Competitor '{competitor name}' already exists with
CompetitorID '{place id}'.")
        return

    # Insert the new competitor into the database using parameterized query
    insert_query = """
        INSERT INTO `{}`.CustomerGbpCompetitors.Competitors` (CompetitorID,
CompetitorName)
        VALUES (@place_id, @competitor_name);
    """.format(project_id)

    insert_job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("place_id", "STRING", place_id),
            bigquery.ScalarQueryParameter("competitor name", "STRING",
competitor name),
        ]
    )

```

```
)

insert_job = client.query(insert_query, job_config=insert_job_config)
insert_job.result() # Wait for the query to complete

logging.info(f"Competitor '{competitor_name}' added with CompetitorID
'{place id}'.")
```

Key Changes

- **Parameterized Queries:** Replaced string concatenation with `bigquery.QueryJobConfig` and `bigquery.ScalarQueryParameter` to safely pass parameters into SQL queries.
- **Consistent Handling:** Ensured all references to `CustomerID` and `CompetitorID` are treated as strings in the queries.

Testing

1. **Deploy Updated Code:** Deploy the updated `big_query.py` as part of your Cloud Function package.
2. **Test the Function:** Send a request to the Cloud Function URL and verify the output.
3. **Check BigQuery:** Confirm that queries are executed correctly and competitors are added as expected.
4. **Monitor Logs:** Review logs to ensure the function is operating without errors.

This should resolve the issue with the SQL query and ensure the identifiers are handled correctly. If you encounter further problems, let me know!

```
This looks like it resolved the error we received during the last execution.
After updating the code and running another test, we received another
error. Below is the verbose output from the failed execution: [7:08:39 PM] -
INFO:root:Starting process for project ID: agency-seo-tools-318618
[7:08:39 PM] - WARNING:google.auth._default:No project ID could be
determined. Consider running gcloud config set project or setting the
GOOGLE_CLOUD_PROJECT environment variable [7:08:39 PM] -
INFO:root:Fetching first customer and queries from BigQuery [7:08:40 PM]
- INFO:root:Found customer: [REDACTED]
[REDACTED] [7:08:41 PM] -
INFO:root:Queries found for customer [REDACTED]
[REDACTED]
[REDACTED] ] INFO:root:Processing customer: [REDACTED]
```

```
[REDACTED] with search location: Toronto,Ontario,Canada
INFO:root:Queries to process: [ [REDACTED]
[REDACTED] ] INFO:root:Performing API search
for query: [REDACTED] [7:08:45 PM] - INFO:root:Found competitor:
Prudent Paws with place_id: ChIJR-KpFInL1IkRa6rAXnc4ES4
ERROR:root:An error occurred: invalid literal for int() with base 10: 'ChIJR-
KpFInL1IkRa6rAXnc4ES4'
```



The error message indicates that there is an attempt to convert a `place_id` to an integer, which is incorrect since `place_id` values are strings. Let's make sure that all references to `place_id` in the code are handled as strings and remove any unnecessary type conversions.

Update `main.py`

Ensure that all `place_id` handling is correctly treated as strings without conversion to integers:

python

```
import functions_framework
import json
import seo_api # Import the seo api module
import big_query # Import the big query module
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"
        logging.info(f"Starting process for project ID: {project_id}")

        # Fetch the first customer and their queries
        customer name, search location, queries
        big_query.get first customer with queries(project id)
        logging.info(f"Processing customer: {customer name} with search
        location: {search location}")
        logging.info(f"Queries to process: {queries}")

        # Create a set to hold unique competitor names
        competitor names set()

        # Loop through all queries for the customer
        for query in queries:
            logging.info(f"Performing API search for query: {query}")

            # Perform an API request for each query
            api_response = seo_api.perform_api_maps_search(query,
            search_location)
            items = api_response.get("tasks", [])[0].get("result", [])[
            0].get("items", [])

            # Find top 6 competitors (exclude customer name)
```

```

        for item in items[:6]:
            business_name = item.get("title")
            place_id = item.get("place_id")

            if business_name != customer_name:
                logging.info(f"Found competitor: {business_name} with
place id: {place_id}")
                competitor_names.add(business_name)

            # Add competitor to database if it doesn't already
exist

            big_query.add_competitor_if_not_exists(
                project_id,
                business_name,
                place_id # Pass place id as a string
            )

        # Prepare the output with the customer name and unique competitors
output = {
    "customer_name": customer_name,
    "unique_competitors": list(competitor_names)
}

        logging.info(f"Process completed for customer: {customer_name}")
        logging.info(f"Unique competitors found:
{output['unique_competitors']}")

        # Return the results as JSON
        return json.dumps(output, indent=2), 200, {'Content Type':
'application/json'}

    except Exception as e:
        logging.error(f"An error occurred: {str(e)}")
        return f"An error occurred: {str(e)}", 500

```

Key Changes

- **String Handling:** Ensure `place_id` is treated as a string throughout the code, especially in the function calls and database inserts.
- **Removed Integer Conversion:** Removed any attempt to convert `place_id` to an integer, as it should always be handled as a string.

Testing

1. **Deploy Updated Code:** Deploy the updated `main.py` file as part of your Cloud Function package.

2. **Test the Function:** Send a request to the Cloud Function URL and verify the output.
3. **Check BigQuery:** Ensure competitors are correctly added with `place_id` as `CompetitorID`.
4. **Monitor Logs:** Review logs to confirm the function executes without errors.

This update should resolve the error related to `place_id` handling. If there are further issues, feel free to ask for more help!

```
After the code changes you suggested, the code executed successfully. I will paste the verbose output from the last code execution. There are two things I'm concerned about the warning that "No project ID could be determined". [7:12:05 PM] - INFO:root:Starting process for project ID: agency-seo-tools-318618 [7:12:05 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running gcloud config set project or setting the GOOGLE_CLOUD_PROJECT environment variable [7:12:05 PM] - INFO:root:Fetching first customer and queries from BigQuery [7:12:06 PM] - INFO:root:Found customer: [REDACTED] [7:12:07 PM] - INFO:root:Queries found for customer [REDACTED]: [REDACTED] INFO:root:Processing customer: [REDACTED] with search location: Toronto,Ontario,Canada INFO:root:Queries to process: [REDACTED] INFO:root:Performing API search for query: [REDACTED] [7:12:11 PM] - INFO:root:Found competitor: Prudent Paws with place_id: ChIJR-KpFInL1IkRa6rAXnc4ES4 [7:12:11 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running gcloud config set project or setting the GOOGLE_CLOUD_PROJECT environment variable [7:12:14 PM] - INFO:root:Competitor 'Prudent Paws' added with CompetitorID 'ChIJR-KpFInL1IkRa6rAXnc4ES4'. INFO:root:Found competitor: Atlas Paws Club with place_id: ChIJcffzWUEXUK0RLBq4liSiMGs [7:12:14 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running gcloud config set project or setting the GOOGLE_CLOUD_PROJECT environment variable [7:12:17 PM] - INFO:root:Competitor 'Atlas Paws Club' added with CompetitorID 'ChIJcffzWUEXUK0RLBq4liSiMGs'. INFO:root:Found competitor: Park9 Downtown- Dog Playcare. Spa. Resort with place_id: ChIJeynt9jzL1IkRozMJHp8pTzY [7:12:17 PM] - WARNING:google.auth._default:No project ID could be determined.
```


Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:12:20 PM] -
INFO:root:Competitor 'Park9 Downtown- Dog Playcare. Spa. Resort' added with CompetitorID 'ChIJeynt9jzL1IkRozMJHp8pTzY'.
INFO:root:Found competitor: SuperDoggos Toronto with place_id: ChIJ2YZVXZQ1K4gRobWTA0Dp6Ss [7:12:20 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:12:23 PM] -
INFO:root:Competitor 'SuperDoggos Toronto' added with CompetitorID 'ChIJ2YZVXZQ1K4gRobWTA0Dp6Ss'. INFO:root:Found competitor: Dog World Resort & Spa with place_id: ChIJOz8XReY1K4gR_tRJmf3LYrU [7:12:23 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:12:26 PM] -
INFO:root:Competitor 'Dog World Resort & Spa' added with CompetitorID 'ChIJOz8XReY1K4gR_tRJmf3LYrU'. INFO:root:Found competitor: Fluffy Paws Pet Care with place_id: ChIJZ0xDkJE1K4gR2M96fxyNM0s [7:12:26 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:12:29 PM] -
INFO:root:Competitor 'Fluffy Paws Pet Care' added with CompetitorID 'ChIJZ0xDkJE1K4gR2M96fxyNM0s'. INFO:root:Performing API search for query: dog boarding toronto [7:12:32 PM] -
INFO:root:Found competitor: Prudent Paws with place_id: ChIJR-KpFlnL1IkRa6rAXnc4ES4 [7:12:32 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:12:33 PM] -
INFO:root:Competitor 'Prudent Paws' already exists with CompetitorID 'ChIJR-KpFlnL1IkRa6rAXnc4ES4'. INFO:root:Found competitor: Atlas Paws Club with place_id: ChIJcffzWUExUK0RLBq4liSiMGs [7:12:33 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:12:34 PM] -
INFO:root:Competitor 'Atlas Paws Club' already exists with CompetitorID 'ChIJcffzWUExUK0RLBq4liSiMGs'. INFO:root:Found competitor: Fluffy Paws Pet Care with place_id: ChIJZ0xDkJE1K4gR2M96fxyNM0s [7:12:34 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:12:35 PM] -
INFO:root:Competitor 'Fluffy Paws Pet Care' already exists with CompetitorID 'ChIJZ0xDkJE1K4gR2M96fxyNM0s'. INFO:root:Found

```
competitor: Park9 Downtown- Dog Playcare. Spa. Resort with place_id:
ChIJeynt9jzL1IkRozMJHp8pTzY [7:12:35 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running gcloud config set project or setting the
GOOGLE_CLOUD_PROJECT environment variable [7:12:36 PM] -
INFO:root:Competitor 'Park9 Downtown- Dog Playcare. Spa. Resort'
already exists with CompetitorID 'ChIJeynt9jzL1IkRozMJHp8pTzY'.
INFO:root:Found competitor: Paws Playgrounds with place_id:
ChIJOaUqA3XL1IkRc314PBuuQiQ [7:12:36 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running gcloud config set project or setting the
GOOGLE_CLOUD_PROJECT environment variable [7:12:39 PM] -
INFO:root:Competitor 'Paws Playgrounds' added with CompetitorID
'ChIJOaUqA3XL1IkRc314PBuuQiQ'. INFO:root:Found competitor:
SuperDoggos Toronto with place_id: ChIJ2YZVXZQ1K4gRobWTA0Dp6Ss
[7:12:39 PM] - WARNING:google.auth._default:No project ID could be
determined. Consider running gcloud config set project or setting the
GOOGLE_CLOUD_PROJECT environment variable [7:12:40 PM] -
INFO:root:Competitor 'SuperDoggos Toronto' already exists with
CompetitorID 'ChIJ2YZVXZQ1K4gRobWTA0Dp6Ss'. INFO:root:Performing
API search for query: dog boarding [7:12:43 PM] - INFO:root:Found
competitor: Black Dog Pet Resort with place_id: None [7:12:43 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running gcloud config set project or setting the
GOOGLE_CLOUD_PROJECT environment variable [7:12:45 PM] -
ERROR:root:An error occurred: 400 Required field CompetitorID cannot
be null Location: US Job ID: 47770fe0-b826-494f-889f-4d42eb66e0c8
```



The warning about the project ID and the error regarding a `None place id` can both be addressed with some adjustments to your setup and code logic.

Handling the Project ID Warning

The warning `No project ID could be determined` indicates that the environment is not aware of your Google Cloud project ID. This is typically set in the environment where your code is running. Here's how to address it:

Set the Project ID in Your Environment

You have two main options to resolve this warning:

1. Set the Project ID using `gcloud` CLI:

If you're using the Google Cloud Shell or any local environment, you can set the project ID using the `gcloud` command:

```
bash
```

```
gcloud config set project agency-seo-tools-318618
```

2. Set the `GOOGLE_CLOUD_PROJECT` Environment Variable:

In your deployment environment (e.g., Google Cloud Functions), you can set environment variables directly in the console or via the deployment configuration:

- Go to your Cloud Function settings in the Google Cloud Console.
- Under "Environment variables," set the key `GOOGLE_CLOUD_PROJECT` with the value `agency-seo-tools-318618`.

Handling the `None place_id`

The error regarding a `None place_id` indicates that the API response might have included a result without a `place_id`. You need to ensure that your code checks for a valid `place_id` before attempting to add a competitor.

Update `main.py` to Check for `place_id`

Here's how you can add a check to ensure the `place_id` is not `None`:

python

```
import functions_framework
import json
import seo_api # Import the seo_api module
import big_query # Import the big_query module
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"
        logging.info(f"Starting process for project ID: {project_id}")

        # Fetch the first customer and their queries
        customer_name, search_location, queries =
big_query.get_first_customer_with_queries(project_id)
        logging.info(f"Processing customer: {customer_name} with search
location: {search_location}")
        logging.info(f"Queries to process: {queries}")

        # Create a set to hold unique competitor names
        competitor_names = set()

        # Loop through all queries for the customer
        for query in queries:
            logging.info(f"Performing API search for query: {query}")

            # Perform an API request for each query
            api_response = seo_api.perform_api_maps_search(query,
search_location)
            items = api_response.get("tasks", [])[0].get("result", [])[
0].get("items", [])

            # Find top 6 competitors (exclude customer name)
```

```

    for item in items[:6]:
        business_name = item.get("title")
        place_id = item.get("place_id")

        # Ensure place_id is not None
        if business_name != customer_name and place_id is not None:
            logging.info(f"Found competitor: {business_name} with
place_id: {place_id}")
            competitor_names.add(business_name)

        # Add competitor to database if it doesn't already
exist
        big_query.add_competitor_if_not_exists(
            project_id,
            business_name,
            place_id # Pass place_id as a string
        )

    # Prepare the output with the customer name and unique competitors
    output = {
        "customer_name": customer_name,
        "unique_competitors": list(competitor_names)
    }

    logging.info(f"Process completed for customer: {customer_name}")
    logging.info(f"Unique competitors found:
{output['unique_competitors']}")

    # Return the results as JSON
    return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}

except Exception as e:
    logging.error(f"An error occurred: {str(e)}")
    return f"An error occurred: {str(e)}", 500

```

Testing

1. **Set Environment Variable:** Ensure that your environment has the `GOOGLE_CLOUD_PROJECT` variable set.
2. **Deploy Updated Code:** Deploy the updated `main.py` file.
3. **Test the Function:** Send a request to the Cloud Function URL and verify the output.
4. **Check BigQuery:** Ensure competitors are correctly added with valid `place_id`.
5. **Monitor Logs:** Review logs to confirm the function executes without warnings and errors.

These updates should resolve both the project ID warning and the error related to `None place_id`. If you encounter any further issues, let me know!

Can we update the `main.py` code so that if the `place_id` is not available that we output a log entry that says that the `place_id` was not found?



Certainly! To log an entry whenever a `place_id` is not available, you can add a conditional check and log a message accordingly. Here's how you can update the `main.py` to include this logging:

Updated `main.py` Code

python

```
import functions_framework
import json
import seo_api # Import the seo_api module
import big_query # Import the big_query module
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"
        logging.info(f"Starting process for project ID: {project_id}")

        # Fetch the first customer and their queries
        customer_name, search_location, queries =
big_query.get_first_customer_with_queries(project_id)
        logging.info(f"Processing customer: {customer_name} with search
location: {search_location}")
        logging.info(f"Queries to process: {queries}")

        # Create a set to hold unique competitor names
        competitor_names = set()

        # Loop through all queries for the customer
        for query in queries:
            logging.info(f"Performing API search for query: {query}")

            # Perform an API request for each query
            api_response = seo_api.perform_api_maps_search(query,
search_location)
            items = api_response.get("tasks", [])[0].get("result", [])[
0].get("items", [])

            # Find top 6 competitors (exclude customer name)
```

```

    for item in items[:6]:
        business_name = item.get("title")
        place_id = item.get("place_id")

        # Check if place_id is available
        if place_id is not None:
            if business_name != customer_name:
                logging.info(f"Found competitor: {business_name}
with place_id: {place_id}")
                competitor_names.add(business_name)

                # Add competitor to database if it doesn't already
exist

                big_query.add_competitor_if_not_exists(
                    project_id,
                    business_name,
                    place_id # Pass place_id as a string
                )
            else:
                logging.warning(f"Place ID not found for competitor:
{business_name}")

        # Prepare the output with the customer name and unique competitors
output = {
    "customer_name": customer_name,
    "unique_competitors": list(competitor_names)
}

        logging.info(f"Process completed for customer: {customer_name}")
        logging.info(f"Unique competitors found:
{output['unique_competitors']}")

        # Return the results as JSON
        return json.dumps(output, indent=2), 200, {'Content-Type':
'application/json'}

    except Exception as e:
        logging.error(f"An error occurred: {str(e)}")
        return f"An error occurred: {str(e)}", 500

```

Key Changes

Logging for Missing `place_id`: Added a logging statement under the `else` clause to capture when a `place_id` is not found for a competitor. This outputs a warning message with the competitor's name.

Testing

1. **Deploy Updated Code:** Ensure the updated `main.py` file is deployed.
2. **Test the Function:** Send a request to the Cloud Function URL and verify the logs.
3. **Check Logs:** Review the logs to confirm that entries are made for competitors with missing `place_id`.

This update will help you track instances where the `place_id` is missing, providing more insight into the data processing flow. If you have further questions or need additional assistance, feel free to ask!

```
The place_id updates look like they work well. The environment variable issue is still unresolved. What I tried was to edit the Cloud Functions project, in the configuration screen there is a section titled "Runtime, build, connections and security settings". I expanded that section and in the Runtime settings I found a 'Runtime environment variables' section where I can set runtime variables. I added the variable "GOOGLE_CLOUD_PROJECT" and set that value to "agency-seo-tools-318618". But this didn't resolve the error. I still see this line during execution: [7:43:59 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running gcloud config set project or setting the GOOGLE_CLOUD_PROJECT environment variable Here is the verbose output from the last execution: [7:43:59 PM] - INFO:root:Starting process for project ID: agency-seo-tools-318618 [7:43:59 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running gcloud config set project or setting the GOOGLE_CLOUD_PROJECT environment variable INFO:root:Fetching first customer and queries from BigQuery [7:44:00 PM] - INFO:root:Found customer: [REDACTED] [7:44:01 PM] - INFO:root:Queries found for customer [REDACTED]: [REDACTED] INFO:root:Processing customer: [REDACTED] with search location: Toronto,Ontario,Canada INFO:root:Queries to process: [REDACTED] INFO:root:Performing API search for query: [REDACTED] [7:44:08 PM] - INFO:root:Found competitor: Fluffy Paws Pet Care with place_id: ChIJZ0xDkJE1K4gR2M96fxyNM0s [7:44:08 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running gcloud config set project or setting the GOOGLE_CLOUD_PROJECT environment variable [7:44:09 PM] - INFO:root:Competitor 'Fluffy Paws Pet Care' already exists with CompetitorID 'ChIJZ0xDkJE1K4gR2M96fxyNM0s'. INFO:root:Found competitor: Prudent Paws with place_id: ChIJR-KpFInL1kRa6rAXnc4ES4
```

[7:44:09 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:44:09 PM] - INFO:root:Competitor 'Prudent Paws' already exists with CompetitorID 'ChIJR-KpFlnL1IkRa6rAXnc4ES4'. INFO:root:Found competitor: Atlas Paws Club with place_id: ChIJcffzWUEXUK0RLBq4liSiMGs [7:44:09 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:44:10 PM] - INFO:root:Competitor 'Atlas Paws Club' already exists with CompetitorID 'ChIJcffzWUEXUK0RLBq4liSiMGs'. INFO:root:Found competitor: Park9 Downtown- Dog Playcare. Spa. Resort with place_id: ChIJeynt9jzL1IkRozMJHp8pTzY [7:44:10 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:44:11 PM] - INFO:root:Competitor 'Park9 Downtown- Dog Playcare. Spa. Resort' already exists with CompetitorID 'ChIJeynt9jzL1IkRozMJHp8pTzY'. INFO:root:Found competitor: K9 Clubhouse with place_id: ChIJRyiJ-5vL1IkRtTNMQq6kDoA [7:44:11 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:44:14 PM] - INFO:root:Competitor 'K9 Clubhouse' added with CompetitorID 'ChIJRyiJ-5vL1IkRtTNMQq6kDoA'. INFO:root:Found competitor: Paula Gilroy with place_id: ChIJeyCjCEzL1IkRAux7ffH4JYE [7:44:14 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:44:17 PM] - INFO:root:Competitor 'Paula Gilroy' added with CompetitorID 'ChIJeyCjCEzL1IkRAux7ffH4JYE'. INFO:root:Performing API search for query: dog boarding toronto [7:44:19 PM] - INFO:root:Found competitor: Prudent Paws with place_id: ChIJR-KpFlnL1IkRa6rAXnc4ES4 [7:44:19 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:44:20 PM] - INFO:root:Competitor 'Prudent Paws' already exists with CompetitorID 'ChIJR-KpFlnL1IkRa6rAXnc4ES4'. INFO:root:Found competitor: Atlas Paws Club with place_id: ChIJcffzWUEXUK0RLBq4liSiMGs [7:44:20 PM] - WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the `GOOGLE_CLOUD_PROJECT` environment variable [7:44:21 PM] - INFO:root:Competitor 'Atlas Paws Club' already exists with CompetitorID

```
'ChIJcfffzWUEXUK0RLBq4liSiMGs'. INFO:root:Found competitor: Fluffy
Paws Pet Care with place_id: ChIJZ0xDkJE1K4gR2M96fxyNM0s [7:44:21
PM] - WARNING:google.auth._default:No project ID could be determined.
Consider running gcloud config set project or setting the
GOOGLE_CLOUD_PROJECT environment variable [7:44:22 PM] -
INFO:root:Competitor 'Fluffy Paws Pet Care' already exists with
CompetitorID 'ChIJZ0xDkJE1K4gR2M96fxyNM0s'. INFO:root:Found
competitor: Paws Playgrounds with place_id:
ChIJOaUqA3XL1IkRc314PBuuQiQ [7:44:22 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running gcloud config set project or setting the
GOOGLE_CLOUD_PROJECT environment variable [7:44:23 PM] -
INFO:root:Competitor 'Paws Playgrounds' already exists with CompetitorID
'ChIJOaUqA3XL1IkRc314PBuuQiQ'. INFO:root:Found competitor: Park9
Downtown- Dog Playcare. Spa. Resort with place_id:
ChIJeynt9jzL1IkRozMJHp8pTzY [7:44:23 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running gcloud config set project or setting the
GOOGLE_CLOUD_PROJECT environment variable [7:44:24 PM] -
INFO:root:Competitor 'Park9 Downtown- Dog Playcare. Spa. Resort'
already exists with CompetitorID 'ChIJeynt9jzL1IkRozMJHp8pTzY'.
INFO:root:Found competitor: K9 Clubhouse with place_id: ChIJRyiJ-
5vL1IkRtTNMQq6kDoA [7:44:24 PM] - WARNING:google.auth._default:No
project ID could be determined. Consider running gcloud config set
project or setting the GOOGLE_CLOUD_PROJECT environment
variable [7:44:25 PM] - INFO:root:Competitor 'K9 Clubhouse' already
exists with CompetitorID 'ChIJRyiJ-5vL1IkRtTNMQq6kDoA'.
INFO:root:Performing API search for query: dog boarding
```



The warning about the project ID is typically caused by the Google Cloud SDK not being able to determine the project ID from the environment in which your code is running. Since you have already set the `GOOGLE_CLOUD_PROJECT` environment variable in the Cloud Functions configuration, the issue may be related to how the BigQuery client is being initialized or how authentication is being handled.

Steps to Resolve the Warning

1. Ensure the Environment Variable is Set Correctly:

- Double-check that the environment variable `GOOGLE_CLOUD_PROJECT` is set to the correct project ID (`agency-seo-tools-318618`) in the Cloud Functions console.
- Ensure there are no typos or extra spaces in the variable name or value.

2. Explicitly Set the Project ID in Code:

You can explicitly set the project ID when creating the BigQuery client to ensure it uses the correct project.

Here's how you can update your code to set the project ID explicitly:

Update `big_query.py`

Modify the `get_bigquery_client` function to use the project ID from the environment variable:

python

```
from google.cloud import bigquery
import os
import logging

def get_bigquery_client():
    """Returns a BigQuery client using the specified project ID."""
    project_id = os.environ.get("GOOGLE_CLOUD_PROJECT", "agency-seo-tools-318618")
    logging.info(f"Using project ID: {project_id}")
    return bigquery.Client(project=project_id)

def get_first_customer_with_queries(project_id):
    """Fetches the first customer and their queries from BigQuery."""
    client = get_bigquery_client()
    logging.info("Fetching first customer and queries from BigQuery")

    # Fetch the first customer with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`
        LIMIT 1;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customer = next(customer_query_job.result(), None) # Get the first customer

    if not customer:
        raise Exception("No customers found in the database.")

    customer_id = customer.get("CustomerID")
    customer_name = customer.get("CustomerName")
    search_location = customer.get("SearchLocation")

    logging.info(f"Found customer: {customer_name} with ID: {customer_id}")

    # Fetch all queries for the customer using parameterized query
    queries_query = """
        SELECT Query
        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
    """
```

```

        WHERE CustomerID = @customer_id;
        """.format(project_id)

    job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("customer_id", "STRING",
customer_id),
        ]
    )

    queries_job = client.query(queries_query, job_config=job_config)
    queries = [row.get("Query") for row in queries_job.result()]

    logging.info(f"Queries found for customer {customer_name}: {queries}")

    return customer_name, search_location, queries

def add_competitor_if_not_exists(project_id, competitor_name, place_id):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client()

    # Check if the competitor already exists using place_id with
parameterized query
    check_query = """
        SELECT CompetitorID
        FROM `{}`.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = @place_id;
        """.format(project_id)

    check_job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("place_id", "STRING", place_id),
        ]
    )

    check_job = client.query(check_query, job_config=check_job_config)
    competitor = next(check_job.result(), None)

    if competitor:
        logging.info(f"Competitor '{competitor_name}' already exists with
CompetitorID '{place_id}'.")
        return

    # Insert the new competitor into the database using parameterized query
    insert_query = """
        INSERT INTO `{}`.CustomerGbpCompetitors.Competitors` (CompetitorID,
CompetitorName)
        VALUES (@place_id, @competitor_name);
        """.format(project_id)

    insert_job_config = bigquery.QueryJobConfig(
        query_parameters=[

```

```

        bigquery.ScalarQueryParameter("place_id", "STRING", place_id),
        bigquery.ScalarQueryParameter("competitor_name", "STRING",
competitor_name),
    ]
)

insert_job = client.query(insert_query, job_config=insert_job_config)
insert_job.result() # Wait for the query to complete

logging.info(f"Competitor '{competitor_name}' added with CompetitorID
'{place_id}'.")

```

Explanation

- **Explicit Project ID:** The `get_bigquery_client` function now retrieves the project ID from the environment variable using `os.environ.get("GOOGLE_CLOUD_PROJECT", "agency-seo-tools-318618")`. This ensures that the project ID is used consistently throughout your application.
- **Fallback Default:** A fallback default project ID is provided in case the environment variable is not set, but it's best to ensure the environment variable is properly set.

Testing

1. **Deploy Updated Code:** Ensure the updated `big_query.py` file is deployed.
2. **Test the Function:** Send a request to the Cloud Function URL and verify the logs.
3. **Check Logs:** Confirm that the warning about the project ID is resolved and that the logs correctly display the project ID being used.

This approach should resolve the project ID warning by ensuring the project ID is correctly set in your environment. If there are further issues, let me know!

Let's move on. Let's update the project code files to expand the functionality. Currently the scripts get the first customer, loop through all queries for that customer and consolidates all top competitors for all queries. Now, let's update the code so that the script performs these operations for all customers in the database.



To update the project code so that the script processes all customers in the database, you'll need to modify the code to loop through each customer and perform the necessary operations for each one. Let's implement these changes in the code.

Step 1: Update `big_query.py`

First, we'll need to update the `get_first_customer_with_queries` function to fetch all customers instead of just the first one.

Updated `big_query.py`

python

```
from google.cloud import bigquery
import os
import logging

def get_bigquery_client():
    """Returns a BigQuery client using the specified project ID."""
    project_id = os.environ.get("GOOGLE_CLOUD_PROJECT", "agency-seo-tools-318618")
    logging.info(f"Using project ID: {project_id}")
    return bigquery.Client(project=project_id)

def get_all_customers_with_queries(project_id):
    """Fetches all customers and their queries from BigQuery."""
    client = get_bigquery_client()
    logging.info("Fetching all customers and queries from BigQuery")

    # Fetch all customers with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customers = customer_query_job.result()

    # Prepare a list to hold customers and their queries
    customers_with_queries = []

    for customer in customers:
        customer_id = customer.get("CustomerID")
        customer_name = customer.get("CustomerName")
        search_location = customer.get("SearchLocation")

        logging.info(f"Found customer: {customer_name} with ID: {customer_id}")

        # Fetch all queries for the customer using parameterized query
        queries_query = """
            SELECT Query
            FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        """
```

```

        WHERE CustomerID = @customer_id;
        """.format(project_id)

    job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("customer_id", "STRING",
customer_id),
        ]
    )

    queries_job = client.query(queries_query, job_config=job_config)
    queries = [row.get("Query") for row in queries_job.result()]

    logging.info(f"Queries found for customer {customer_name}:
{queries}")

    # Add the customer and their queries to the list
    customers_with_queries.append({
        "customer_id": customer_id,
        "customer_name": customer_name,
        "search_location": search_location,
        "queries": queries
    })

    return customers_with_queries

def add_competitor_if_not_exists(project_id, competitor_name, place_id):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client()

    # Check if the competitor already exists using place_id with
parameterized query
    check_query = """
        SELECT CompetitorID
        FROM `{}`.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = @place_id;
        """.format(project_id)

    check_job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("place_id", "STRING", place_id),
        ]
    )

    check_job = client.query(check_query, job_config=check_job_config)
    competitor = next(check_job.result(), None)

    if competitor:
        logging.info(f"Competitor '{competitor_name}' already exists with
CompetitorID '{place_id}'.")
        return

```

```

# Insert the new competitor into the database using parameterized query
insert_query = """
    INSERT INTO `{}`.CustomerGbpCompetitors.Competitors` (CompetitorID,
CompetitorName)
    VALUES (@place_id, @competitor_name);
""".format(project_id)

insert_job_config = bigquery.QueryJobConfig(
    query_parameters=[
        bigquery.ScalarQueryParameter("place_id", "STRING", place_id),
        bigquery.ScalarQueryParameter("competitor_name", "STRING",
competitor_name),
    ]
)

insert_job = client.query(insert_query, job_config=insert_job_config)
insert_job.result() # Wait for the query to complete

logging.info(f"Competitor '{competitor_name}' added with CompetitorID
'{place_id}'.")

```

Step 2: Update `main.py`

Next, update `main.py` to loop through each customer and process their queries:

python

```
import functions_framework
import json
import seo_api # Import the seo_api module
import big_query # Import the big_query module
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"
        logging.info(f"Starting process for project ID: {project_id}")

        # Fetch all customers and their queries
        customers_with_queries =
big_query.get_all_customers_with_queries(project_id)

        # Process each customer
        for customer_data in customers_with_queries:
            customer_name = customer_data["customer_name"]
            search_location = customer_data["search_location"]
            queries = customer_data["queries"]

            logging.info(f"Processing customer: {customer_name} with search
location: {search_location}")
            logging.info(f"Queries to process: {queries}")

            # Create a set to hold unique competitor names
            competitor_names = set()

            # Loop through all queries for the customer
            for query in queries:
                logging.info(f"Performing API search for query: {query}")
```

```

        # Perform an API request for each query
        api_response = seo_api.perform_api_maps_search(query,
search_location)
        items = api_response.get("tasks", [])[0].get("result", [])[0].get("items", [])

        # Find top 6 competitors (exclude customer name)
        for item in items[:6]:
            business_name = item.get("title")
            place_id = item.get("place_id")

            # Check if place_id is available
            if place_id is not None:
                if business_name != customer_name:
                    logging.info(f"Found competitor:
{business_name} with place_id: {place_id}")
                    competitor_names.add(business_name)

            # Add competitor to database if it doesn't
already exist

            big_query.add_competitor_if_not_exists(
                project_id,
                business_name,
                place_id # Pass place_id as a string
            )
        else:
            logging.warning(f"Place ID not found for
competitor: {business_name}")

        # Log the unique competitors for the current customer
        logging.info(f"Unique competitors found for customer
{customer_name}: {list(competitor_names)}")

        logging.info("Process completed for all customers.")

        # Return a success message
        return "Process completed successfully for all customers.", 200,
{'Content-Type': 'text/plain'}

    except Exception as e:
        logging.error(f"An error occurred: {str(e)}")
        return f"An error occurred: {str(e)}", 500

```

Explanation

- **get_all_customers_with_queries**: This function now retrieves all customers and their associated queries from BigQuery.

- **Loop Through Customers:** The main function loops through each customer, processes their queries, and collects competitors.
- **Logging:** Detailed logging is included to track progress and any issues encountered.

Testing

1. **Deploy Updated Code:** Deploy the updated `main.py` and `big_query.py` files as part of your Cloud Function package.
2. **Test the Function:** Send a request to the Cloud Function URL and verify the output.
3. **Check Logs:** Review logs to ensure that all customers are processed and any issues are logged.

These updates will enable the script to process all customers in the database and consolidate competitors for each one. Let me know if you need further adjustments or have additional questions!

This looks like it worked, but there's a problem with the data because two customers have the same `place_id` value in the database. Can you write the SQL to update the `CustomerID` for [REDACTED] to [REDACTED]? You can use the business name to find the record and update it. The business name in the database is "[REDACTED]"



To update the `CustomerID` for "[REDACTED]" in your `Customers` table, you can use the following SQL query. This query will find the record by its business name and update the `CustomerID`:

sql

```
UPDATE `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`
SET CustomerID = '[REDACTED]'
WHERE CustomerName = '[REDACTED]';
```

Steps to Execute the SQL

-
1. **Open BigQuery Console:** Go to the BigQuery section of the Google Cloud Console.
 2. **Select Your Project:** Ensure you have the correct project selected.
 3. **Run the Query:**
 - o Open the SQL workspace.
 - o Copy and paste the above SQL query into the query editor.
 - o Execute the query to update the record.

Verification

After running the query, you can verify that the update was successful by executing a SELECT query:

```
sql
```

```
SELECT *
FROM `agency-seo-tools-318618.CustomerGbpCompetitors.Customers`
WHERE CustomerName = '[REDACTED]';
```

This will show you the updated record to confirm that the **CustomerID** has been changed as intended.

Considerations

- Ensure that no other records have duplicate **CustomerID** values.
- Double-check that the change aligns with the actual **place_id** for "Wiebe Criminal Defence."

These steps will correct the **CustomerID** for the specified business in your database. Let me know if you encounter any issues or need further assistance!

Now we have to update CustomerID to [REDACTED] in the CustomerQueries table for records who have a QueryID value of 7 or 8



To update the **CustomerID** in the **CustomerQueries** table for records with specific **QueryID** values, you can use the following SQL query. This query will update the **CustomerID** for the specified **QueryID** values:

sql

```
UPDATE `agency-seo-tools-318618.CustomerGbpCompetitors.CustomerQueries`  
SET CustomerID = '████████████████████'  
WHERE QueryID IN (7, 8);
```

Steps to Execute the SQL

1. **Open BigQuery Console:** Go to the BigQuery section of the Google Cloud Console.
2. **Select Your Project:** Ensure you have the correct project selected.
3. **Run the Query:**
 - Open the SQL workspace.
 - Copy and paste the above SQL query into the query editor.
 - Execute the query to update the records.

Verification

After running the query, you can verify that the update was successful by executing a **SELECT** query:

sql

```
SELECT *  
FROM `agency-seo-tools-318618.CustomerGbpCompetitors.CustomerQueries`  
WHERE QueryID IN (7, 8);
```


This will show you the updated records to confirm that the `CustomerID` has been changed as intended.

Considerations

- Ensure that the updated `CustomerID` matches the `place_id` for "Wiebe Criminal Defence" as expected.
- Double-check that no other records are unintentionally affected by this change.

These steps will update the `CustomerID` for the specified queries in your database. If you need further assistance, feel free to ask!

Okay, looks like the problem may have been resolved now. Let's move on. We forgot to include one step during the process of finding all unique competitors across all queries for each customer. During this process, we have to update the `CustomersCompetitors` table with the `CustomerID` and `CompetitorID` for each unique competitor found.



To update the `CustomersCompetitors` table with the `CustomerID` and `CompetitorID` for each unique competitor found during the process, you need to add an additional step in your code. This step will involve checking if the association between a customer and a competitor already exists and inserting a new record if it doesn't.

Step 1: Update `big_query.py`

We'll add a function to handle the insertion of records into the `CustomersCompetitors` table.

Updated `big_query.py`

python

```
from google.cloud import bigquery
import os
import logging

def get_bigquery_client():
    """Returns a BigQuery client using the specified project ID."""
    project_id = os.environ.get("GOOGLE_CLOUD_PROJECT", "agency-seo-tools-318618")
    logging.info(f"Using project ID: {project_id}")
    return bigquery.Client(project=project_id)

def get_all_customers_with_queries(project_id):
    """Fetches all customers and their queries from BigQuery."""
    client = get_bigquery_client()
    logging.info("Fetching all customers and queries from BigQuery")

    # Fetch all customers with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customers = customer_query_job.result()

    # Prepare a list to hold customers and their queries
    customers_with_queries = []

    for customer in customers:
        customer_id = customer.get("CustomerID")
        customer_name = customer.get("CustomerName")
        search_location = customer.get("SearchLocation")

        logging.info(f"Found customer: {customer_name} with ID: {customer_id}")

        # Fetch all queries for the customer using parameterized query
        queries_query = """
            SELECT Query
            FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        """
```

```

        WHERE CustomerID = @customer_id;
        """.format(project_id)

    job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("customer_id", "STRING",
customer_id),
        ]
    )

    queries_job = client.query(queries_query, job_config=job_config)
    queries = [row.get("Query") for row in queries_job.result()]

    logging.info(f"Queries found for customer {customer_name}:
{queries}")

    # Add the customer and their queries to the list
    customers_with_queries.append({
        "customer_id": customer_id,
        "customer_name": customer_name,
        "search_location": search_location,
        "queries": queries
    })

    return customers_with_queries

def add_competitor_if_not_exists(project_id, competitor_name, place_id):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client()

    # Check if the competitor already exists using place_id with
parameterized query
    check_query = """
        SELECT CompetitorID
        FROM `{}`.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = @place_id;
        """.format(project_id)

    check_job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("place_id", "STRING", place_id),
        ]
    )

    check_job = client.query(check_query, job_config=check_job_config)
    competitor = next(check_job.result(), None)

    if competitor:
        logging.info(f"Competitor '{competitor_name}' already exists with
CompetitorID '{place_id}'.")
        return

```

```

# Insert the new competitor into the database using parameterized query
insert_query = """
    INSERT INTO `{}`.CustomerGbpCompetitors.Competitors` (CompetitorID,
CompetitorName)
    VALUES (@place_id, @competitor_name);
""".format(project_id)

insert_job_config = bigquery.QueryJobConfig(
    query_parameters=[
        bigquery.ScalarQueryParameter("place_id", "STRING", place_id),
        bigquery.ScalarQueryParameter("competitor_name", "STRING",
competitor_name),
    ]
)

insert_job = client.query(insert_query, job_config=insert_job_config)
insert_job.result() # Wait for the query to complete

logging.info(f"Competitor '{competitor_name}' added with CompetitorID
'{{place_id}}'.")

def add_customer_competitor_association(project_id, customer_id,
competitor_id):
    """Adds an association between a customer and a competitor if it
doesn't exist."""
    client = get_bigquery_client()

    # Check if the association already exists
    check_query = """
        SELECT 1
        FROM `{}`.CustomerGbpCompetitors.CustomersCompetitors`
        WHERE CustomerID = @customer_id AND CompetitorID = @competitor_id;
""".format(project_id)

    check_job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("customer_id", "STRING",
customer_id),
            bigquery.ScalarQueryParameter("competitor_id", "STRING",
competitor_id),
        ]
    )

    check_job = client.query(check_query, job_config=check_job_config)
    association_exists = next(check_job.result(), None)

    if association_exists:
        logging.info(f"Association between CustomerID '{customer_id}' and
CompetitorID '{competitor_id}' already exists.")
        return

    # Insert the new association into the database

```

```

insert_query = """
    INSERT INTO `{}`.CustomerGbpCompetitors.CustomersCompetitors`
(CustomerID, CompetitorID)
    VALUES (@customer_id, @competitor_id);
""".format(project_id)

insert_job_config = bigquery.QueryJobConfig(
    query_parameters=[
        bigquery.ScalarQueryParameter("customer_id", "STRING",
customer_id),
        bigquery.ScalarQueryParameter("competitor_id", "STRING",
competitor_id),
    ]
)

insert_job = client.query(insert_query, job_config=insert_job_config)
insert_job.result() # Wait for the query to complete

logging.info(f"Association between CustomerID '{customer_id}' and
CompetitorID '{competitor_id}' added.")

```

Step 2: Update `main.py`

Modify the code to update the `CustomersCompetitors` table with each unique competitor found:

python

```
import functions_framework
import json
import seo_api # Import the seo_api module
import big_query # Import the big_query module
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"
        logging.info(f"Starting process for project ID: {project_id}")

        # Fetch all customers and their queries
        customers_with_queries =
big_query.get_all_customers_with_queries(project_id)

        # Process each customer
        for customer_data in customers_with_queries:
            customer_id = customer_data["customer_id"]
            customer_name = customer_data["customer_name"]
            search_location = customer_data["search_location"]
            queries = customer_data["queries"]

            logging.info(f"Processing customer: {customer_name} with search
location: {search_location}")
            logging.info(f"Queries to process: {queries}")

            # Create a set to hold unique competitor names
            competitor_names = set()

            # Loop through all queries for the customer
            for query in queries:
                logging.info(f"Performing API search for query: {query}")
```

```

        # Perform an API request for each query
        api_response = seo_api.perform_api_maps_search(query,
search_location)
        items = api_response.get("tasks", [])[0].get("result", [])[0].get("items", [])

        # Find top 6 competitors (exclude customer name)
        for item in items[:6]:
            business_name = item.get("title")
            place_id = item.get("place_id")

            # Check if place_id is available
            if place_id is not None:
                if business_name != customer_name:
                    logging.info(f"Found competitor:
{business_name} with place_id: {place_id}")
                    competitor_names.add(business_name)

                    # Add competitor to database if it doesn't
already exist

                    big_query.add_competitor_if_not_exists(
                        project_id,
                        business_name,
                        place_id # Pass place_id as a string
                    )

                    # Add association between customer and
competitor

                    big_query.add_customer_competitor_association(
                        project_id,
                        customer_id,
                        place_id
                    )

                else:
                    logging.warning(f"Place ID not found for
competitor: {business_name}")

            # Log the unique competitors for the current customer
            logging.info(f"Unique competitors found for customer
{customer_name}: {list(competitor_names)}")

        logging.info("Process completed for all customers.")

        # Return a success message
        return "Process completed successfully for all customers.", 200,
{'Content-Type': 'text/plain'}

    except Exception as e:
        logging.error(f"An error occurred: {str(e)}")
        return f"An error occurred: {str(e)}", 500

```

Explanation

- **add_customer_competitor_association:** This function checks if an association between a customer and a competitor already exists in the `CustomersCompetitors` table and inserts a new record if it doesn't.
- **Main Processing Loop:** After adding a competitor, the code now also adds the association between the customer and the competitor

Great. Now let's move on. Let's add the functionality of populating the data in the `BusinessData` table: - RecordID (Integer, Required) - RecordDate (Date, Required) - BusinessID (String, Required) - BusinessType (String, Required) - BusinessName (String, Required) - ReviewCount (Integer, Nullable) - AverageRating (Float, Nullable) RecordID should be an incrementing number starting from 1. The script should look for the largest RecordID and then increment that by one to add a new record. The BusinessID is the `place_id` for the Customer or Competitor. The BusinessType should either be "Customer" or "Competitor" The BusinessName, ReviewCount, and AverageRating should be retrieved from the API's output. The code should populate the `BusinessData` table with the Customer's data as well as all unique competitors. Below is sample output from the API to use for determining where to retrieve values from.

```
{ "id": "08090332-8148-0139-0000-b62fd6410371", "status_code": 20000, "status_message": "Ok.", "time": "4.0533 sec.", "cost": 0.002, "result_count": 1, "path": [ "v3", "serp", "google", "maps", "live", "advanced" ], "data": { "api": "serp", "function": "live", "se": "google", "se_type": "maps", "keyword": "commercial cleaning company", "location_code": 1002200, "language_code": "en", "device": "desktop", "os": "windows", "depth": 10 }, "result": [ { "keyword": "commercial cleaning company", "type": "maps", "se_domain": "google.ca", "location_code": 1002200, "language_code": "en", "check_url": "https://google.ca/maps/search/commercial+cleaning+company/@43.3616211,-80.3hl=en&gl=CA&uule=w+CAIQIFISCRIs5CC4iSulEXAdI8coewMF", "datetime": "2024-08-09 00:33:00 +00:00", "spell": null, "item_types": [ "maps_search" ], "se_results_count": 0, "items_count": 10, "items": [ { "type": "maps_search", "rank_group": 1, "rank_absolute": 1, "domain": "kitchenercleaningservice.ca", "title": "KCS Kitchener Cleaning Services", "url": "https://kitchenercleaningservice.ca/?utm_source=gmb&utm_medium=referral", "contact_url": "https://kitchenercleaningservice.ca/?utm_source=organic&utm_medium=gmb&utm_campaign=appointment_link", "rating": { "rating_type": "Max5", "value": 5, "votes_count": 54,
```



```
"rating_max": null }, "hotel_rating": null, "price_level": null,
"rating_distribution": { "1": 0, "2": 0, "3": 0, "4": 0, "5": 54 }, "snippet": "135
Hardcastle Dr, Cambridge, ON N1S 0A6", "address": "135 Hardcastle Dr,
Cambridge, ON N1S 0A6", "address_info": { "borough": null, "address":
"135 Hardcastle Dr", "city": "Cambridge", "zip": "N1S 0A6", "region":
"Ontario", "country_code": "CA" }, "place_id":
"ChIJSZm_ZIhUNUkR3gbDymYFA0U", "phone": "+1226-400-7376",
"main_image":
"https://lh5.googleusercontent.com/p/AF1QipMgTrGi8LhvC61OO2nul43v1hZovAplp
h306-k-no", "total_photos": 89, "category": "House cleaning service",
"additional_categories": [ "Commercial cleaning service" ], "category_ids": [
"house_cleaning_service", "commercial_cleaning_service" ], "work_hours":
{ "timetable": { "sunday": null, "monday": [ { "open": { "hour": 9, "minute": 0
}, "close": { "hour": 22, "minute": 0 } } ], "tuesday": [ { "open": { "hour": 9,
"minute": 0 }, "close": { "hour": 22, "minute": 0 } } ], "wednesday": [ { "open":
{ "hour": 9, "minute": 0 }, "close": { "hour": 22, "minute": 0 } } ], "thursday": [
{ "open": { "hour": 9, "minute": 0 }, "close": { "hour": 22, "minute": 0 } } ],
"friday": [ { "open": { "hour": 9, "minute": 0 }, "close": { "hour": 22, "minute":
0 } } ], "saturday": [ { "open": { "hour": 9, "minute": 0 }, "close": { "hour": 18,
"minute": 0 } } ] } ], "current_status": "open" }, "feature_id":
"0x4935545866bf9949:0x45030566cac306de", "cid":
"4972824352593741534", "latitude": 43.3530636, "longitude":
-80.3462157, "is_claimed": true, "local_justifications": null,
"is_directory_item": false }, { "type": "maps_search", "rank_group": 2,
"rank_absolute": 2, "domain": "www.silvermapleservices.com", "title":
"Silver Maple Maintenance", "url": "http://www.silvermapleservices.com/",
"contact_url": null, "rating": { "rating_type": "Max5", "value": 5,
"votes_count": 10, "rating_max": null }, "hotel_rating": null, "price_level":
null, "rating_distribution": { "1": 0, "2": 0, "3": 0, "4": 0, "5": 10 }, "snippet":
"70 Dudhope Ave, Cambridge, ON N1R 4T6", "address": "70 Dudhope
Ave, Cambridge, ON N1R 4T6", "address_info": { "borough": "Christopher-
Champlain", "address": "70 Dudhope Ave", "city": "Cambridge", "zip": "N1R
4T6", "region": "Ontario", "country_code": "CA" }, "place_id":
"ChIJURatUr13LlgRgfU0_2bmjd8", "phone": "+1226-791-7517",
"main_image":
"https://lh5.googleusercontent.com/p/AF1QipNY2ZAKg0ojrGvma_HcgRCQqGjIAHrC
h272-k-no", "total_photos": 4, "category": "Commercial cleaning service",
"additional_categories": null, "category_ids": [
"commercial_cleaning_service" ], "work_hours": { "timetable": { "sunday": [
{ "open": { "hour": 8, "minute": 0 }, "close": { "hour": 19, "minute": 0 } } ],
"monday": [ { "open": { "hour": 8, "minute": 0 }, "close": { "hour": 19,
"minute": 0 } } ], "tuesday": [ { "open": { "hour": 8, "minute": 0 }, "close": {
```

```
"hour": 19, "minute": 0 } } ], "wednesday": [ { "open": { "hour": 8, "minute": 0
}, "close": { "hour": 19, "minute": 0 } } ], "thursday": [ { "open": { "hour": 8,
"minute": 0 }, "close": { "hour": 19, "minute": 0 } } ], "friday": [ { "open": {
"hour": 8, "minute": 0 }, "close": { "hour": 19, "minute": 0 } } ], "saturday": [ {
"open": { "hour": 8, "minute": 0 }, "close": { "hour": 19, "minute": 0 } } ] ],
"current_status": "close" }, "feature_id":
"0x882c77bd52ad1651:0xdf8de666ff34f581", "cid":
"16108784772216845697", "latitude": 43.349884599999996, "longitude":
-80.300219099999999, "is_claimed": true, "local_justifications": null,
"is_directory_item": false }, { "type": "maps_search", "rank_group": 3,
"rank_absolute": 3, "domain": null, "title": "Stratus Building Solutions", "url":
null, "contact_url": null, "rating": { "rating_type": "Max5", "value": 5,
"votes_count": 4, "rating_max": null }, "hotel_rating": null, "price_level":
null, "rating_distribution": { "1": 0, "2": 0, "3": 0, "4": 0, "5": 4 }, "snippet":
"653 Franklin Blvd Unit 2, Cambridge, ON N1R 7Z1", "address": "653
Franklin Blvd Unit 2, Cambridge, ON N1R 7Z1", "address_info": {
"borough": "Fiddlesticks", "address": "653 Franklin Blvd Unit 2", "city":
"Cambridge", "zip": "N1R 7Z1", "region": "Ontario", "country_code": "CA" },
"place_id": "ChIJid4ZIYX1K4gRDWh3D0u009Y", "phone": "+1519-404-
7417", "main_image":
"https://lh5.googleusercontent.com/p/AF1QipPSLryp7WKxgc2JBWI98T3DQLDAxE5
h544-k-no", "total_photos": 4, "category": "Commercial cleaning service",
"additional_categories": [ "Cleaners", "Janitorial service" ], "category_ids": [
"commercial_cleaning_service", "cleaners", "janitorial_service" ],
"work_hours": { "timetable": { "sunday": null, "monday": [ { "open": { "hour":
8, "minute": 30 }, "close": { "hour": 17, "minute": 0 } } ], "tuesday": [ { "open":
{ "hour": 8, "minute": 30 }, "close": { "hour": 17, "minute": 0 } } ],
"wednesday": [ { "open": { "hour": 8, "minute": 30 }, "close": { "hour": 17,
"minute": 0 } } ], "thursday": [ { "open": { "hour": 8, "minute": 30 }, "close": {
"hour": 17, "minute": 0 } } ], "friday": [ { "open": { "hour": 8, "minute": 30 },
"close": { "hour": 17, "minute": 0 } } ], "saturday": null }, "current_status":
"close" }, "feature_id": "0x882bf5852119de89:0xd6d3b44b0f77680d", "cid":
"15479914578677557261", "latitude": 43.376776799999995, "longitude":
-80.2971543, "is_claimed": true, "local_justifications": null,
"is_directory_item": false }, { "type": "maps_search", "rank_group": 4,
"rank_absolute": 4, "domain": "www.jan-pro.ca", "title": "JAN-PRO
Kitchener-Waterloo", "url": "https://www.jan-pro.ca/en/office/kitchener-
waterloo/?
utm_source=google&utm_medium=organic&utm_campaign=local&utm_content=Kitc
"contact_url": null, "rating": { "rating_type": "Max5", "value": 4.9,
"votes_count": 21, "rating_max": null }, "hotel_rating": null, "price_level":
null, "rating_distribution": { "1": 0, "2": 0, "3": 0, "4": 2, "5": 19 }, "snippet":
```

"543 Mill St #1, Kitchener, ON N2G 2Y5", "address": "543 Mill St #1, Kitchener, ON N2G 2Y5", "address_info": { "borough": "Rockway", "address": "543 Mill St #1", "city": "Kitchener", "zip": "N2G 2Y5", "region": "Ontario", "country_code": "CA" }, "place_id": "ChIJ5ZDbstaJK4gRq9-dlPAj1oE", "phone": "+1519-342-3688", "main_image": "https://lh5.googleusercontent.com/p/AF1QipNPIOxWflrc7jeq1ryxY_EXATImXlbsjrkp h242-k-no", "total_photos": 3, "category": "Commercial cleaning service", "additional_categories": ["Cleaners", "Janitorial service"], "category_ids": ["commercial_cleaning_service", "cleaners", "janitorial_service"], "work_hours": { "timetable": { "sunday": null, "monday": [{ "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } }], "tuesday": [{ "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } }], "wednesday": [{ "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } }], "thursday": [{ "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } }], "friday": [{ "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } }], "saturday": null }, "current_status": "close" }, "feature_id": "0x882b89d6b2db90e5:0x81d623f0949ddfab", "cid": "9355704792101281707", "latitude": 43.432705999999996, "longitude": -80.47580959999999, "is_claimed": true, "local_justifications": null, "is_directory_item": false }, { "type": "maps_search", "rank_group": 5, "rank_absolute": 5, "domain": null, "title": "MC - Cleaning Services", "url": null, "contact_url": null, "rating": { "rating_type": "Max5", "value": 5, "votes_count": 1, "rating_max": null }, "hotel_rating": null, "price_level": null, "rating_distribution": { "1": 0, "2": 0, "3": 0, "4": 0, "5": 1 }, "snippet": "18 Concession St Unit 305, Cambridge, ON N1R 2G5", "address": "18 Concession St Unit 305, Cambridge, ON N1R 2G5", "address_info": { "borough": null, "address": "18 Concession St Unit 305", "city": "Cambridge", "zip": "N1R 2G5", "region": "Ontario", "country_code": "CA" }, "place_id": "ChIJpz8UIkV3LIgRHbQzmqp0zBQ", "phone": "+1226-476-3364", "main_image": "https://lh5.googleusercontent.com/p/AF1QipOGliS1KMCu5A8UuMIOh9OFzBLI59IF h271-k-no", "total_photos": 4, "category": "Janitorial service", "additional_categories": ["Cleaners", "Commercial cleaning service", "House cleaning service", "Window cleaning service"], "category_ids": ["janitorial_service", "cleaners", "commercial_cleaning_service", "house_cleaning_service", "window_cleaning_service"], "work_hours": { "timetable": { "sunday": [{ "open": { "hour": 7, "minute": 0 }, "close": { "hour": 19, "minute": 0 } }], "monday": [{ "open": { "hour": 7, "minute": 0 }, "close": { "hour": 19, "minute": 0 } }], "tuesday": [{ "open": { "hour": 7, "minute": 0 }, "close": { "hour": 19, "minute": 0 } }], "wednesday": [{ "open": { "hour": 7, "minute": 0 }, "close": { "hour": 19, "minute": 0 } }], "thursday": [{ "open": { "hour": 7, "minute": 0 }, "close": { "hour": 19, "minute": 0 } }],

```
"friday": [ { "open": { "hour": 7, "minute": 0 }, "close": { "hour": 19, "minute": 0 } } ], "saturday": [ { "open": { "hour": 7, "minute": 0 }, "close": { "hour": 19, "minute": 0 } } ] ], "current_status": "close" }, "feature_id": "0x882c774522143fa7:0x14cc74aa9a33b41d", "cid": "1498701052087874589", "latitude": 43.3536959, "longitude": -80.3145359, "is_claimed": true, "local_justifications": null, "is_directory_item": false }, { "type": "maps_search", "rank_group": 6, "rank_absolute": 6, "domain": "www.kitchenerclean.com", "title": "Kitchener Clean", "url": "http://www.kitchenerclean.com/", "contact_url": null, "rating": { "rating_type": "Max5", "value": 4.9, "votes_count": 15, "rating_max": null }, "hotel_rating": null, "price_level": null, "rating_distribution": { "1": 0, "2": 0, "3": 0, "4": 2, "5": 13 }, "snippet": "565 Landgren Ct, Kitchener, ON N2A 0G8", "address": "565 Landgren Ct, Kitchener, ON N2A 0G8", "address_info": { "borough": "Grand River South", "address": "565 Landgren Ct", "city": "Kitchener", "zip": "N2A 0G8", "region": "Ontario", "country_code": "CA" }, "place_id": "ChIJ63OD6tkLLIgRwEqM-g64LzE", "phone": "+1519-513-3965", "main_image": "https://lh5.googleusercontent.com/p/AF1QipPIHRQjtfDXXdD39LDsssL5TdMU_0Prz6pBm=w408-h272-k-no", "total_photos": 4, "category": "Commercial cleaning service", "additional_categories": null, "category_ids": [ "commercial_cleaning_service" ], "work_hours": { "timetable": { "sunday": null, "monday": [ { "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } } ], "tuesday": [ { "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } } ], "wednesday": [ { "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } } ], "thursday": [ { "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } } ], "friday": [ { "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } } ], "saturday": null }, "current_status": "close" }, "feature_id": "0x882c0bd9ea8373eb:0x312fb80efa8c4ac0", "cid": "3544253806236420800", "latitude": 43.4459805, "longitude": -80.4119746, "is_claimed": true, "local_justifications": null, "is_directory_item": false }, { "type": "maps_search", "rank_group": 7, "rank_absolute": 7, "domain": "www.georgesjanitorial.ca", "title": "George's Janitorial Services", "url": "https://www.georgesjanitorial.ca/", "contact_url": null, "rating": { "rating_type": "Max5", "value": 5, "votes_count": 13, "rating_max": null }, "hotel_rating": null, "price_level": null, "rating_distribution": { "1": 0, "2": 0, "3": 0, "4": 0, "5": 13 }, "snippet": null, "address": null, "address_info": { "borough": null, "address": null, "city": null, "zip": null, "region": null, "country_code": null }, "place_id": "ChIJt4Y_I993LIgRKMVu63kOn5Y", "phone": "+1647-261-2267", "main_image": "https://lh5.googleusercontent.com/p/AF1QipPTV3IzPz2uJlpvk1WGitsMWzIBb-
```

```
skyigH5KCx=w430-h240-k-no", "total_photos": 4, "category": "Commercial
cleaning service", "additional_categories": [ "Service establishment" ],
"category_ids": [ "commercial_cleaning_service", "establishment_service"
], "work_hours": { "timetable": { "sunday": [ { "open": { "hour": 13, "minute":
0 }, "close": { "hour": 18, "minute": 0 } } ], "monday": [ { "open": { "hour": 8,
"minute": 0 }, "close": { "hour": 22, "minute": 0 } } ], "tuesday": [ { "open": {
"hour": 8, "minute": 0 }, "close": { "hour": 22, "minute": 0 } } ], "wednesday":
[ { "open": { "hour": 8, "minute": 0 }, "close": { "hour": 22, "minute": 0 } } ],
"thursday": [ { "open": { "hour": 8, "minute": 0 }, "close": { "hour": 22,
"minute": 0 } } ], "friday": [ { "open": { "hour": 8, "minute": 0 }, "close": {
"hour": 22, "minute": 0 } } ], "saturday": [ { "open": { "hour": 9, "minute": 0 },
"close": { "hour": 18, "minute": 0 } } ] }, "current_status": "open" },
"feature_id": "0x882c77df973f86b7:0x969f0e79eb6ec528", "cid":
"10853409543789921576", "latitude": 43.4321464, "longitude":
-80.4387029, "is_claimed": true, "local_justifications": null,
"is_directory_item": false }, { "type": "maps_search", "rank_group": 8,
"rank_absolute": 8, "domain": null, "title": "Green Team Commercial Clean",
"url": null, "contact_url": null, "rating": { "rating_type": "Max5", "value": 4.8,
"votes_count": 8, "rating_max": null }, "hotel_rating": null, "price_level":
null, "rating_distribution": { "1": 0, "2": 0, "3": 0, "4": 2, "5": 6 }, "snippet":
"120 Carlyle Dr, Kitchener, ON N2P 1N6", "address": "120 Carlyle Dr,
Kitchener, ON N2P 1N6", "address_info": { "borough": "Pioneer Park",
"address": "120 Carlyle Dr", "city": "Kitchener", "zip": "N2P 1N6", "region":
"Ontario", "country_code": "CA" }, "place_id": "ChIJva9mIS31K4gRgxL5M-
M1dWk", "phone": "+1519-722-5326", "main_image":
"https://lh5.googleusercontent.com/p/AF1QipMvrpISvNaal74Z8BJim2_NCXP2ZeAuy
h272-k-no", "total_photos": 3, "category": "Commercial cleaning service",
"additional_categories": null, "category_ids": [
"commercial_cleaning_service" ], "work_hours": { "timetable": { "sunday": [
{ "open": { "hour": 0, "minute": 0 }, "close": { "hour": 0, "minute": 0 } } ],
"monday": [ { "open": { "hour": 0, "minute": 0 }, "close": { "hour": 0, "minute":
0 } } ], "tuesday": [ { "open": { "hour": 0, "minute": 0 }, "close": { "hour": 0,
"minute": 0 } } ], "wednesday": [ { "open": { "hour": 0, "minute": 0 }, "close": {
"hour": 0, "minute": 0 } } ], "thursday": [ { "open": { "hour": 0, "minute": 0 },
"close": { "hour": 0, "minute": 0 } } ], "friday": [ { "open": { "hour": 0, "minute":
0 }, "close": { "hour": 0, "minute": 0 } } ], "saturday": [ { "open": { "hour": 0,
"minute": 0 }, "close": { "hour": 0, "minute": 0 } } ] }, "current_status": "open"
}, "feature_id": "0x882bf52d2166afbd:0x697535e333f91283", "cid":
"7599039196203389571", "latitude": 43.395053, "longitude": -80.4539054,
"is_claimed": true, "local_justifications": null, "is_directory_item": false }, {
"type": "maps_search", "rank_group": 9, "rank_absolute": 9, "domain":
"jdicleaning.com", "title": "JDI Cleaning Services
```

Kitchener/Waterloo/Guelph/Cambridge/Brantford", "url":
"https://jdicleaning.com/commercial-cleaning-services/kitchener-on/",
"contact_url": "https://jdicleaning.com/contact/", "rating": { "rating_type":
"Max5", "value": 4.5, "votes_count": 67, "rating_max": null }, "hotel_rating":
null, "price_level": null, "rating_distribution": { "1": 6, "2": 1, "3": 1, "4": 6,
"5": 53 }, "snippet": "871 Victoria St N Unit 209A, Kitchener, ON N2B 3S4",
"address": "871 Victoria St N Unit 209A, Kitchener, ON N2B 3S4",
"address_info": { "borough": "Rosemount", "address": "871 Victoria St N
Unit 209A", "city": "Kitchener", "zip": "N2B 3S4", "region": "Ontario",
"country_code": "CA" }, "place_id": "ChIJOxxmyvPzK4gRPUsgWAAIkI4",
"phone": "+1519-884-9600", "main_image":
"https://lh5.googleusercontent.com/p/AF1QipP5IWIxzK9DV3d9NcVI7bKEWF46-
jQ33pwq9hg=w408-h408-k-no", "total_photos": 4, "category": "Commercial
cleaning service", "additional_categories": ["Carpet cleaning service",
"Janitorial service", "Sanitation service"], "category_ids": [
"commercial_cleaning_service", "carpet_cleaning_service",
"janitorial_service", "sanitation_service"], "work_hours": { "timetable": {
"sunday": null, "monday": [{ "open": { "hour": 9, "minute": 0 }, "close": {
"hour": 17, "minute": 0 } }], "tuesday": [{ "open": { "hour": 9, "minute": 0 },
"close": { "hour": 17, "minute": 0 } }], "wednesday": [{ "open": { "hour": 9,
"minute": 0 }, "close": { "hour": 17, "minute": 0 } }], "thursday": [{ "open": {
"hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } }], "friday": [{
"open": { "hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } }],
"saturday": null }, "current_status": "close" }, "feature_id":
"0x882bf3f3ca661c3b:0x8e90250058064b3d", "cid":
"10272751433439136573", "latitude": 43.463114499999996, "longitude":
-80.4656346, "is_claimed": true, "local_justifications": null,
"is_directory_item": false }, { "type": "maps_search", "rank_group": 10,
"rank_absolute": 10, "domain": "jvscleaningservices.ca", "title": "JVS
Cleaning Services", "url": "https://jvscleaningservices.ca/", "contact_url":
null, "rating": { "rating_type": "Max5", "value": 5, "votes_count": 1,
"rating_max": null }, "hotel_rating": null, "price_level": null,
"rating_distribution": { "1": 0, "2": 0, "3": 0, "4": 0, "5": 1 }, "snippet": "349
Blair Rd, Cambridge, ON N1S 2K3", "address": "349 Blair Rd, Cambridge,
ON N1S 2K3", "address_info": { "borough": "Blair Road", "address": "349
Blair Rd", "city": "Cambridge", "zip": "N1S 2K3", "region": "Ontario",
"country_code": "CA" }, "place_id": "ChIJ9bAddv53LIgR6e8Wm56b3FU",
"phone": "+1226-336-6857", "main_image":
"https://lh5.googleusercontent.com/p/AF1QipOas-
rNg7iRjh9UI8dBH58k2GwHqQ37q1BalVwc=w408-h544-k-no",
"total_photos": 4, "category": "Commercial cleaning service",
"additional_categories": null, "category_ids": [

```
"commercial_cleaning_service" ], "work_hours": { "timetable": { "sunday":
null, "monday": [ { "open": { "hour": 9, "minute": 0 }, "close": { "hour": 17,
"minute": 0 } } ], "tuesday": [ { "open": { "hour": 9, "minute": 0 }, "close": {
"hour": 17, "minute": 0 } } ], "wednesday": [ { "open": { "hour": 9, "minute": 0
}, "close": { "hour": 17, "minute": 0 } } ], "thursday": [ { "open": { "hour": 9,
"minute": 0 }, "close": { "hour": 17, "minute": 0 } } ], "friday": [ { "open": {
"hour": 9, "minute": 0 }, "close": { "hour": 17, "minute": 0 } } ], "saturday":
null }, "current_status": "close" }, "feature_id":
"0x882c77fe761db0f5:0x55dc9b9e9b16efe9", "cid":
"6186991093609328617", "latitude": 43.374094299999996, "longitude":
-80.338674, "is_claimed": true, "local_justifications": null,
"is_directory_item": false } ] ] ] }
```



To populate the `BusinessData` table with information about both customers and competitors, we need to perform the following steps:

1. Retrieve the largest existing `RecordID` from the `BusinessData` table to determine the next `RecordID`.
2. For each customer and their unique competitors found in the API response, gather the necessary data (`BusinessID`, `BusinessType`, `BusinessName`, `ReviewCount`, and `AverageRating`).
3. Insert this data into the `BusinessData` table, incrementing the `RecordID` for each new record.

Step 1: Update `big_query.py`

Add functions to handle retrieving the maximum `RecordID` and inserting data into the `BusinessData` table.

Updated `big_query.py`

python

```
from google.cloud import bigquery
import os
import logging
from datetime import date

def get_bigquery_client():
    """Returns a BigQuery client using the specified project ID."""
    project_id = os.environ.get("GOOGLE_CLOUD_PROJECT", "agency-seo-tools-318618")
    logging.info(f"Using project ID: {project_id}")
    return bigquery.Client(project=project_id)

def get_all_customers_with_queries(project_id):
    """Fetches all customers and their queries from BigQuery."""
    client = get_bigquery_client()
    logging.info("Fetching all customers and queries from BigQuery")

    # Fetch all customers with their search location
    customer_query = """
        SELECT
            CustomerID,
            CustomerName,
            SearchLocation
        FROM
            `{}`.CustomerGbpCompetitors.Customers`;
    """.format(project_id)

    customer_query_job = client.query(customer_query)
    customers = customer_query_job.result()

    # Prepare a list to hold customers and their queries
    customers_with_queries = []

    for customer in customers:
        customer_id = customer.get("CustomerID")
        customer_name = customer.get("CustomerName")
        search_location = customer.get("SearchLocation")

        logging.info(f"Found customer: {customer_name} with ID: {customer_id}")

        # Fetch all queries for the customer using parameterized query
        queries_query = """
            SELECT Query
```



```

        FROM `{}`.CustomerGbpCompetitors.CustomerQueries`
        WHERE CustomerID = @customer_id;
    """.format(project_id)

    job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("customer_id", "STRING",
customer_id),
        ]
    )

    queries_job = client.query(queries_query, job_config=job_config)
    queries = [row.get("Query") for row in queries_job.result()]

    logging.info(f"Queries found for customer {customer_name}:
{queries}")

    # Add the customer and their queries to the list
    customers_with_queries.append({
        "customer_id": customer_id,
        "customer_name": customer_name,
        "search_location": search_location,
        "queries": queries
    })

    return customers_with_queries

def add_competitor_if_not_exists(project_id, competitor_name, place_id):
    """Adds a competitor to the database if it does not already exist."""
    client = get_bigquery_client()

    # Check if the competitor already exists using place_id with
parameterized query
    check_query = """
        SELECT CompetitorID
        FROM `{}`.CustomerGbpCompetitors.Competitors`
        WHERE CompetitorID = @place_id;
    """.format(project_id)

    check_job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("place_id", "STRING", place_id),
        ]
    )

    check_job = client.query(check_query, job_config=check_job_config)
    competitor = next(check_job.result(), None)

    if competitor:
        logging.info(f"Competitor '{competitor_name}' already exists with
CompetitorID '{place_id}'.")
        return

```

```

# Insert the new competitor into the database using parameterized query
insert_query = """
    INSERT INTO `{}`.CustomerGbpCompetitors.Competitors` (CompetitorID,
CompetitorName)
    VALUES (@place_id, @competitor_name);
""".format(project_id)

insert_job_config = bigquery.QueryJobConfig(
    query_parameters=[
        bigquery.ScalarQueryParameter("place_id", "STRING", place_id),
        bigquery.ScalarQueryParameter("competitor_name", "STRING",
competitor_name),
    ]
)

insert_job = client.query(insert_query, job_config=insert_job_config)
insert_job.result() # Wait for the query to complete

logging.info(f"Competitor '{competitor_name}' added with CompetitorID
'{{place_id}}'.")

def add_customer_competitor_association(project_id, customer_id,
competitor_id):
    """Adds an association between a customer and a competitor if it
doesn't exist."""
    client = get_bigquery_client()

    # Check if the association already exists
    check_query = """
        SELECT 1
        FROM `{}`.CustomerGbpCompetitors.CustomersCompetitors`
        WHERE CustomerID = @customer_id AND CompetitorID = @competitor_id;
    """.format(project_id)

    check_job_config = bigquery.QueryJobConfig(
        query_parameters=[
            bigquery.ScalarQueryParameter("customer_id", "STRING",
customer_id),
            bigquery.ScalarQueryParameter("competitor_id", "STRING",
competitor_id),
        ]
    )

    check_job = client.query(check_query, job_config=check_job_config)
    association_exists = next(check_job.result(), None)

    if association_exists:
        logging.info(f"Association between CustomerID '{customer_id}' and
CompetitorID '{competitor_id}' already exists.")
        return

```

```

# Insert the new association into the database
insert_query = """
    INSERT INTO `{}`.CustomerGbpCompetitors.CustomersCompetitors`
(CustomerID, CompetitorID)
    VALUES (@customer_id, @competitor_id);
""".format(project_id)

insert_job_config = bigquery.QueryJobConfig(
    query_parameters [
        bigquery.ScalarQueryParameter("customer id", "STRING",
customer id),
        bigquery.ScalarQueryParameter("competitor id", "STRING",
competitor id),
    ]
)

insert_job = client.query(insert_query, job_config=insert_job_config)
insert_job.result() # Wait for the query to complete

logging.info(f"Association between CustomerID '{customer_id}' and
CompetitorID '{competitor_id}' added.")

def get_next_record_id(project_id):
    """Fetches the next RecordID for the BusinessData table."""
    client = get_bigquery_client()

    query = """
        SELECT MAX(RecordID) as max_record_id
        FROM `{}`.CustomerGbpCompetitors.BusinessData`;
    """.format(project_id)

    query_job = client.query(query)
    result = next(query_job.result(), None)

    if result and result.max_record_id is not None:
        return result.max_record_id + 1
    return 1

def insert_business_data(project_id, record_id, business_id, business_type,
business_name, review_count, average_rating):
    """Inserts a new record into the BusinessData table."""
    client = get_bigquery_client()

    # Insert the new record using parameterized query
    insert_query = """
        INSERT INTO `{}`.CustomerGbpCompetitors.BusinessData`
        (RecordID, RecordDate, BusinessID, BusinessType, BusinessName,
ReviewCount, AverageRating)
        VALUES (@record_id, @record_date, @business_id, @business_type,
@business_name, @review_count, @average_rating);
    """.format(project_id)

```

```

insert_job_config = bigquery.QueryJobConfig(
    query_parameters=[
        bigquery.ScalarQueryParameter("record_id", "INT64", record_id),
        bigquery.ScalarQueryParameter("record_date", "DATE",
date.today().strftime("%Y-%m-%d")),
        bigquery.ScalarQueryParameter("business_id", "STRING",
business id),
        bigquery.ScalarQueryParameter("business type", "STRING",
business type),
        bigquery.ScalarQueryParameter("business name", "STRING",
business name),
        bigquery.ScalarQueryParameter("review count", "INT64",
review count),
        bigquery.ScalarQueryParameter("average rating", "FLOAT64",
average rating),
    ]
)

insert_job = client.query(insert_query, job_config=insert_job_config)
insert_job.result() # Wait for the query to complete

logging.info(f"Inserted business data for {business_type}
'{business_name}' with RecordID {record_id}.")

```

Step 2: Update `main.py`

Modify the main processing loop to gather data for both customers and competitors and insert it into the `BusinessData` table.

Updated `main.py`

python

```
import functions_framework
import json
import seo_api # Import the seo api module
import big_query # Import the big query module
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"
        logging.info(f"Starting process for project ID: {project_id}")

        # Fetch all customers and their queries
        customers_with_queries = big_query.get_all_customers_with_queries(project_id)

        # Process each customer
        for customer_data in customers_with_queries:
            customer_id = customer_data["customer id"]
            customer_name = customer_data["customer name"]
            search_location = customer_data["search location"]
            queries = customer_data["queries"]

            logging.info(f"Processing customer: {customer_name} with search
            location: {search_location}")
            logging.info(f"Queries to process: {queries}")

            # Create a set to hold unique competitor names
            competitor_names = set()

            # Loop through all queries for the customer
            for query in queries:
                logging.info(f"Performing API search for query: {query}")
```

```

# Perform an API request for each query
api_response = seo_api.perform_api_maps_search(query,
search_location)
items = api_response.get("tasks", [])[0].get("result", [])[0].get("items", [])

# Get the next record ID
next record id    big query.get next record id(project id)

# Find top 6 competitors (exclude customer name)
for item in items[:6]:
    business name    item.get("title")
    place id    item.get("place id")
    rating info    item.get("rating", {})
    review count    rating info.get("votes count", 0)
    average rating    rating info.get("value", 0.0)

# Determine if this is a customer or competitor
if business_name == customer_name:
    business_type = "Customer"
else:
    business_type = "Competitor"

# Add data to the BusinessData table
big_query.insert_business_data(
    project_id,
    next record id,
    place id,
    business type,
    business name,
    review count,
    average rating
)

# Only consider competitors for unique tracking
if business type    "Competitor" and place id is not
None:
    if business_name != customer_name:
        logging.info(f"Found competitor:
{business_name} with place_id: {place_id}")
        competitor_names.add(business_name)

# Add competitor to database if it doesn't
already exist

big_query.add_competitor_if_not_exists(
    project_id,
    business_name,
    place_id # Pass place_id as a string
)

# Add association between customer and

```

```

competitor

        big_query.add_customer_competitor_association(
            project_id,
            customer_id,
            place_id
        )

        next_record_id + 1

    else:
        logging.warning(f"Place ID not found for
competitor: {business name}")

        # Log the unique competitors for the current customer
        logging.info(f"Unique competitors found for customer
{customer name}: {list(competitor names)}")

        logging.info("Process completed for all customers.")

        # Return a success message
        return "Process completed successfully for all customers.", 200,
{'Content-Type': 'text/plain'}

except Exception as e:
    logging.error(f"An error occurred: {str(e)}")
    return f"An error occurred: {str(e)}", 500

```

Explanation

- **get_next_record_id**: This function retrieves the highest **RecordID** in the **BusinessData** table and calculates the next available **RecordID**.
- **insert business data**: Inserts a new record into the **BusinessData** table with the details from the API response.
- **Main Processing Loop**: The loop now gathers data for both the customer and competitors, updating the **BusinessData** table accordingly. The loop also ensures **RecordID** is incremented for each new entry.

This setup will ensure that the **BusinessData** table is populated with a historical record of both customers' and competitors' data, enabling you to track changes over time.

I updated the project with the code you provided, but we ran into an error:
"ERROR:root:An error occurred: 'NoneType' object has no attribute 'get'"
Below is the verbose output from the last execution: [8:49:37 PM] -
INFO:root:Starting process for project ID: agency-seo-tools-318618
INFO:root:Using project ID: agency-seo-tools-318618 [8:49:37 PM] -

WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the
GOOGLE_CLOUD_PROJECT environment variable [8:49:37 PM] -
INFO:root:Fetching all customers and queries from BigQuery [8:49:38 PM]
- INFO:root:Found customer: KCS Kitchener Cleaning Services with ID:
ChIJSZm_ZlhUNUKR3gbDymYFA0U [8:49:39 PM] - INFO:root:Queries
found for customer KCS Kitchener Cleaning Services: ['commercial
cleaning cambridge', 'commercial cleaning company', 'house cleaning
cambridge', 'commercial cleaning', 'house cleaners cambridge']
INFO:root:Found customer: [REDACTED]
[REDACTED] [8:49:40 PM] -
INFO:root:Queries found for customer [REDACTED]
[REDACTED]: [REDACTED]
[REDACTED]'] INFO:root:Found customer:
[REDACTED] with ID: [REDACTED]
[8:49:41 PM] - INFO:root:Queries found for customer [REDACTED]
[REDACTED]: [REDACTED]
[REDACTED]'] INFO:root:Found customer:
[REDACTED] with ID: [REDACTED] [8:49:41
PM] - INFO:root:Queries found for customer [REDACTED]
[REDACTED]: [REDACTED]
[REDACTED]'] INFO:root:Processing customer: KCS Kitchener
Cleaning Services with search location: Kitchener,Ontario,Canada
INFO:root:Queries to process: ['commercial cleaning cambridge',
'commercial cleaning company', 'house cleaning cambridge', 'commercial
cleaning', 'house cleaners cambridge'] INFO:root:Performing API search
for query: commercial cleaning cambridge [8:49:45 PM] - INFO:root:Using
project ID: agency-seo-tools-318618 [8:49:45 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the
GOOGLE_CLOUD_PROJECT environment variable [8:49:46 PM] -
INFO:root:Using project ID: agency-seo-tools-318618 [8:49:46 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the
GOOGLE_CLOUD_PROJECT environment variable [8:49:48 PM] -
INFO:root:Inserted business data for Competitor 'Tripple J Cleaning' with
RecordID 1. INFO:root:Found competitor: Tripple J Cleaning with place_id:
ChIJ2Ro85i2JK4gR-hJeT8SG9lg INFO:root:Using project ID: agency-seo-
tools-318618 [8:49:48 PM] - WARNING:google.auth._default:No project ID
could be determined. Consider running `gcloud config set project` or
setting the GOOGLE_CLOUD_PROJECT environment variable [8:49:49

PM] - INFO:root:Competitor 'Tripple J Cleaning' already exists with CompetitorID 'ChIJ2Ro85i2JK4gR-hJeT8SG9Ig'. INFO:root:Using project ID: agency-seo-tools-318618 [8:49:49 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:49:50 PM] -
INFO:root:Association between CustomerID 'ChIJSZm_ZlhUNUkR3gbDymYFA0U' and CompetitorID 'ChIJ2Ro85i2JK4gR-hJeT8SG9Ig' already exists. INFO:root:Using project ID: agency-seo-tools-318618 [8:49:50 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:49:52 PM] -
INFO:root:Inserted business data for Competitor 'JAN-PRO Kitchener-Waterloo' with RecordID 2. INFO:root:Found competitor: JAN-PRO Kitchener-Waterloo with place_id: ChIJ5ZDbstaJK4gRq9-dIPAj1oE
INFO:root:Using project ID: agency-seo-tools-318618 [8:49:52 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:49:53 PM] -
INFO:root:Competitor 'JAN-PRO Kitchener-Waterloo' already exists with CompetitorID 'ChIJ5ZDbstaJK4gRq9-dIPAj1oE'. INFO:root:Using project ID: agency-seo-tools-318618 [8:49:53 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:49:54 PM] -
INFO:root:Association between CustomerID 'ChIJSZm_ZlhUNUkR3gbDymYFA0U' and CompetitorID 'ChIJ5ZDbstaJK4gRq9-dIPAj1oE' already exists. INFO:root:Using project ID: agency-seo-tools-318618 [8:49:54 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:49:56 PM] -
INFO:root:Inserted business data for Competitor 'JDI Cleaning Services Kitchener/Waterloo/Guelph/Cambridge/Brantford' with RecordID 3.
INFO:root:Found competitor: JDI Cleaning Services Kitchener/Waterloo/Guelph/Cambridge/Brantford with place_id: ChIJOxxmyvPzK4gRPUsgWAAIkI4 INFO:root:Using project ID: agency-seo-tools-318618 [8:49:56 PM] -
WARNING:google.auth._default:No project ID could be determined. Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:49:57 PM] -
INFO:root:Competitor 'JDI Cleaning Services

Kitchener/Waterloo/Guelph/Cambridge/Brantford' already exists with CompetitorID 'ChIJOxxmyvPzK4gRPUsgWAAIkI4'. INFO:root:Using project ID: agency-seo-tools-318618 [8:49:57 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:49:58 PM] -
INFO:root:Association between CustomerID 'ChIJSZm_ZlhUNUkR3gbDymYFA0U' and CompetitorID 'ChIJOxxmyvPzK4gRPUsgWAAIkI4' already exists. INFO:root:Using project ID: agency-seo-tools-318618 [8:49:58 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:50:00 PM] -
INFO:root:Inserted business data for Competitor 'Lemac Maintenance' with RecordID 4. INFO:root:Found competitor: Lemac Maintenance with place_id: ChIJ99oh-nSiFIQRaGSheD7E0B0 INFO:root:Using project ID: agency-seo-tools-318618 [8:50:00 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:50:01 PM] -
INFO:root:Competitor 'Lemac Maintenance' already exists with CompetitorID 'ChIJ99oh-nSiFIQRaGSheD7E0B0'. INFO:root:Using project ID: agency-seo-tools-318618 [8:50:01 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:50:02 PM] -
INFO:root:Association between CustomerID 'ChIJSZm_ZlhUNUkR3gbDymYFA0U' and CompetitorID 'ChIJ99oh-nSiFIQRaGSheD7E0B0' already exists. INFO:root:Using project ID: agency-seo-tools-318618 [8:50:02 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:50:04 PM] -
INFO:root:Inserted business data for Competitor 'M&H CLEANING SERVICES LTD.' with RecordID 5. INFO:root:Found competitor: M&H CLEANING SERVICES LTD. with place_id: ChIJK8e8QVx9PGoRw_yWy9-4_DQ INFO:root:Using project ID: agency-seo-tools-318618 [8:50:04 PM] -
WARNING:google.auth._default:No project ID could be determined.
Consider running `gcloud config set project` or setting the GOOGLE_CLOUD_PROJECT environment variable [8:50:05 PM] -
INFO:root:Competitor 'M&H CLEANING SERVICES LTD.' already exists with CompetitorID 'ChIJK8e8QVx9PGoRw_yWy9-4_DQ'. INFO:root:Using

```
project ID: agency-seo-tools-318618 [8:50:05 PM] -  
WARNING:google.auth._default:No project ID could be determined.  
Consider running gcloud config set project or setting the  
GOOGLE_CLOUD_PROJECT environment variable [8:50:06 PM] -  
INFO:root:Association between CustomerID  
'ChIJSZm_ZlhUNUkR3gbDymYFA0U' and CompetitorID  
'ChIJK8e8QVx9PGoRw_yWy9-4_DQ' already exists. ERROR:root:An error  
occurred: 'NoneType' object has no attribute 'get'
```



The error message `'NoneType' object has no attribute 'get'` suggests that somewhere in your code, you are trying to call the `.get()` method on a `None` object. This likely happens when the API response is not structured as expected, and the code tries to access a key that doesn't exist.

Troubleshooting Steps

1. **Check API Response Structure:** Ensure that the API response is structured as expected and contains the necessary data fields.
2. **Add Defensive Code:** Implement checks to handle situations where expected keys are missing in the response. This can be done by checking if a key exists before accessing it.
3. **Improve Logging:** Add logging statements to capture the API response and identify where the structure might differ from what the code expects.

Updated Code with Defensive Checks

Let's modify the relevant section in the `main.py` file to add defensive checks and logging:

Updated `main.py`

python

```
import functions_framework
import json
import seo_api # Import the seo api module
import big_query # Import the big query module
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)

@functions_framework.http
def script_init(request):
    """HTTP Cloud Function.
    Args:
        request (flask.Request): The request object.
    Returns:
        The response text, or any set of values that can be turned into a
        Response object using `make_response`.
    """
    try:
        # Project ID for BigQuery
        project_id = "agency-seo-tools-318618"
        logging.info(f"Starting process for project ID: {project_id}")

        # Fetch all customers and their queries
        customers_with_queries = big_query.get_all_customers_with_queries(project_id)

        # Process each customer
        for customer_data in customers_with_queries:
            customer_id = customer_data["customer id"]
            customer_name = customer_data["customer name"]
            search_location = customer_data["search location"]
            queries = customer_data["queries"]

            logging.info(f"Processing customer: {customer_name} with search
            location: {search_location}")
            logging.info(f"Queries to process: {queries}")

            # Create a set to hold unique competitor names
            competitor_names = set()

            # Loop through all queries for the customer
            for query in queries:
                logging.info(f"Performing API search for query: {query}")
```

```

# Perform an API request for each query
api_response = seo_api.perform_api_maps_search(query,
search_location)
tasks = api_response.get("tasks", [])

# Ensure the task and result exist
if not tasks or not tasks[0].get("result"):
    logging.warning(f"No results found for query: {query}")
    continue

items = tasks[0]["result"][0].get("items", [])

# Get the next record ID
next_record_id = big_query.get_next_record_id(project_id)

# Find top 6 competitors (exclude customer name)
for item in items[:6]:
    business_name = item.get("title", "Unknown")
    place_id = item.get("place_id")

    # Check if rating info is present
    rating_info = item.get("rating", {})
    review_count = rating_info.get("votes_count", 0)
    average_rating = rating_info.get("value", 0.0)

    # Determine if this is a customer or competitor
    if business_name == customer_name:
        business_type = "Customer"
    else:
        business_type = "Competitor"

    if place_id is None:
        logging.warning(f"Place ID not found for business:
{business_name}")
        continue

    # Add data to the BusinessData table
    big_query.insert_business_data(
        project_id,
        next_record_id,
        place_id,
        business_type,
        business_name,
        review_count,
        average_rating
    )

# Only consider competitors for unique tracking
if business_type == "Competitor":
    if business_name != customer_name:
        logging.info(f"Found competitor:

```

```

{business_name} with place_id: {place_id}")
        competitor_names.add(business_name)

        # Add competitor to database if it doesn't
already exist

        big_query.add_competitor_if_not_exists(
            project id,
            business name,
            place id # Pass place id as a string
        )

        # Add association between customer and
competitor

        big query.add customer competitor association(
            project id,
            customer id,
            place id
        )

        next_record_id += 1

        # Log the unique competitors for the current customer
        logging.info(f"Unique competitors found for customer
{customer_name}: {list(competitor_names)}")

        logging.info("Process completed for all customers.")

        # Return a success message
        return "Process completed successfully for all customers.", 200,
{'Content Type': 'text/plain'}

    except Exception as e:
        logging.error(f"An error occurred: {str(e)}")
        return f"An error occurred: {str(e)}", 500

```

Key Changes

- 1. Defensive Checks:** Added checks to ensure that the `tasks` list and `result` field are not empty before attempting to access them. If they are missing, a warning is logged and the loop continues.
- 2. Logging Improvements:** Added logging to capture situations where no results are found or a `place id` is missing.
- 3. Default Values:** Used default values when calling `.get()` on dictionary items to prevent `NoneType` errors.

By implementing these changes, the code should be more robust and handle unexpected API responses more gracefully. Let me know if you encounter any other issues or need further adjustments!

Can you write a BigQuery SQL query that will do the following: Get all `BusinessData` records for KCS Kitchener Cleaning Services (CustomerID: `ChIJSZm_ZlhUNUKR3gbDymYFA0U`) and all Competitors that are in the `CustomerCompetitors` table that have the CustomerID set to KCS's ID?



To retrieve all `BusinessData` records for "KCS Kitchener Cleaning Services" and its competitors from BigQuery, you can use a `JOIN` operation between the `BusinessData` table and the `CustomersCompetitors` table to match records based on the `CustomerID`. The query will look like this:

sql

```
SELECT bd.*
FROM `your_project_id.CustomerGbpCompetitors.BusinessData` bd
JOIN `your_project_id.CustomerGbpCompetitors.CustomersCompetitors` cc
  ON bd.BusinessID = cc.CompetitorID
WHERE cc.CustomerID = 'ChIJSZm_ZlhUNUKR3gbDymYFA0U'
      OR bd.BusinessID = 'ChIJSZm_ZlhUNUKR3gbDymYFA0U'
```

Explanation

- **SELECT bd.*:** This selects all columns from the `BusinessData` table.
- **JOIN operation:** This joins the `BusinessData` table (`bd`) with the `CustomersCompetitors` table (`cc`) on the condition that the `BusinessID` in `BusinessData` matches the `CompetitorID` in `CustomersCompetitors`.
- **WHERE clause:**
 - Retrieves records where the `CustomerID` in the `CustomersCompetitors` table matches the ID of "KCS Kitchener Cleaning Services."
 - Additionally, it includes records from `BusinessData` where the `BusinessID` matches "KCS Kitchener Cleaning Services" to get its data as well.

Replace `your_project_id` with your actual Google Cloud project ID.

This query should return all the records for KCS Kitchener Cleaning Services and its competitors as defined in your `CustomersCompetitors` table. Let me know if you need further assistance or any modifications to this query!

ChatGPT can make mistakes. Check important info.